IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

Applicant:	Robert M. Craig	Examiner:	Baoquoc N. To
Serial No.:	09/360,440	Group Art Unit:	2784
Filed:	July 26, 1999	Docket No.:	MS140696.1/40062.25US01
Title:	LOGIC TABLE ABSTRACTION LAYER FOR ACCESSING CONFIGURATION INFORMATION		

CERTIFICATE UNDER 37 CFR 1.10

'Express Mail' mailing label number: EV 118156791 US
Date of Deposit: June 20 2005

I hereby certify that this paper or fee is being deposited with the United States Postal Service 'Express Mail Post Office To Addressee' service under 37 CFR 1.10 on the date indicated above and is addressed to: Mail Stop Amendment, Commissioner of Patents, P.O. Box 1450, Alexandria, VA 22313-1450.

By: Jennifer Weck
Name: JENTFER WECK

EXHIBIT A
DECLARATION UNDER 37 CFR § 1.131

I, Robert M. Craig, declared as follows:

1. I am the sole named inventor on U.S. Patent Application Serial No. 09/360,440, filed July 26, 1999 (hereinafter "this application").

2. I am aware that an Office Action was mailed in this application on February 24, 2005, and that, in this Office Action, all pending claims were rejected either as being anticipated by U.S. Patent No. 6,487,552 (hereinafter Lei) under 35 U.S.C. § 102(e), or as being unpatentable over Lei in view of U.S. Patent No. 5,581,765 (Monroe).

3. The invention set forth in all claims submitted in this application, whether pending, original or previously amended, was conceived and disclosed on or before July 18, 1997. As shown in Exhibit B, I provided a detailed email on July 18, 1997 that described the present invention as defined in the claims. Attached to the email was a draft specification and draft presentation, attached hereto as Exhibit C and Exhibit D, which provided more detailed description of the instant invention. The email, specification, and presentation show the aspects of the invention including the conception of the table objects and interactions between the table objects as defined in the claims. In particular, the sections entitled "How To Obtain A Table,"

"How Data Tables Work," and "How Logic Tables Work" describe many of the aspects of the invention defined in the claims.

4. A working embodiment of the invention was incorporated into and shipped with the Windows NT 5.0 Beta version 2.0. As shown in the Microsoft Press Release entitled "Company Focuses on Quality and Customer Feedback," dated August 18, 1998, working version of the Windows NT 5.0 Beta version 2.0 was released to beta testers on or before August 18, 1998. See Exhibit E.

5. I herby declare that all statements made herein of my own knowledge are true and that all statements are made with the knowledge that willful false statements and the like so made are punishable by fine or imprisonment, or both, under Section 1001 of Title 18 of the United States Code and that such false statements may jeopardize the validity of the application or any patent issued thereon.

Date: 6/20/2005

Robert M. Craig

Tadd F. Wilson

From: Tim Scull
Sent: Thursday, April 21, 2005 10:09 AM
To: Tadd F. Wilson
Subject: FW: Application Server Patent meeting recap


catinfra.doc


comcat.ppt

-----Original Message-----
From: Homer Knearl
Sent: Friday, June 18, 1999 9:58 AM
To: Richard Holzer; Tim Scull; Richard Gregson; Amy Xu
Cc: 'kemew@microsoft.com'; Betty Terry
Subject: FW: Application Server Patent meeting recap

Rick, Tim, Richard and Amy:

Attached is the information for disclosure review meetings. We have a specification and a powerpoint presentation for the product. We will find out the direction for each disclosure at the meetings. In the meantime you should review the spec and the presentation before the meetings.

I have assigned you as follows:

Monday 9-12 AM: MS#140695.1 and MS#140697.1 -- Tim Scull Monday 12:15 - 5 PM: MS# 140698.1 and MS#140700.1 -- Rich Gregson Tuesday 9 Am - 1:15 PM: MS# 140701.1 and MS# 140696.1 -- Rick Holzer Tuesday 3 - 5 PM: MS# 140699.1 -- Amy Xu

Amy, hopefully the meeting with Eric Watson re UEI will be between 1:15 - 3 PM.

Regarding the attachments I was able to open the work document with Word 95. The Powerpoint document had to be opened with PPT 98.

Adjust your reservations to fit your schedule. Thanks.

Homer

-----Original Message-----
From: Keme Green (Woods LCA) [mailto:kemew@microsoft.com]
Sent: Friday, June 18, 1999 9:48 AM
To: 'hknearl@merchant-gould.com'
Cc: Janice Francisco (LCA); Keme Green (Woods LCA); LCA Records Center; Heather Garcia (LCA); Victoria Savacool (LCA)
Subject: FW: Application Server Patent meeting recap

Hi Homer, I am forwarding the attachments below containing specification information for the disclosure meetings that you're scheduled to attend on Monday and Tuesday next week (June 21-22, 1999).

Please let me know if you need any further assistance.

Thank you,
Keme Green
Paralegal Assistantl

> Here is the original first draft "spec" of the simple table based catalog,

> sent about 2 years ago (I have emails for some months prior but this is
> the first formal write-up): nicely enough, it describes the problems I was
> originally trying to solve:
> <<MTS3 Catalog Infrastructure Specification>>
>
>
> Other documents of interest can be found at \\rcraig0\public (I can dig up
> more emails, etc too as necessary):
> *   catinfra.doc: The original (and unfortunately incomplete spec).
> Pretty dense reading but details problems, logic tables, etc.
> *   comcat.ppt: A slide deck (run it, it uses lots of builds)
> pictographically detailing the catalog as it appears in COM+ 1.0.
> *   simpletablesv2.doc: The new v2 spec, in progress. A lot of this is
> just documenting what we did in COM+ 1.0. Appendix C details changes from
> v1.
>
> <<catinfra.doc>> <<comcat.ppt>>
>
> Also at \\rcraig0\public:
> *   stable.idl.old: The IDL for simple tables v1 (from COM+ 1.0 source).
> *   stable.idl.new: The IDL for simple tables v2 (from Catalog42 source,
> in progress).
>
>
> Thanks,
> Robert Craig.
>
>

Message-ID: <301478D07C83D011807200805F5019570129B23B@red-20-msg.dns.microsoft.com>
From: Robert Craig <rcraig@microsoft.com>
To: Bill Devlin <billdev@microsoft.com>, Markus Horstmann
<markush@microsoft.com>, Michael Nelson <t-miken@microsoft.com>,
Wenjun Qiu <wenjunq@microsoft.com>, Wilf Russell <wilfr@microsoft.com>,
Cheng Zhou <t-chengz@microsoft.com>, Jeff Miller <jeffmill@MICROSOFT.com>
Cc: Rodney Limprecht <rodneyl@microsoft.com>, Greg Hope
<greghope@microsoft.com>, Dave Reed <davereed@microsoft.com>
Subject: MTS3 Catalog Infrastructure Specification
Date: Fri, 18 Jul 1997 16:13:07 -0700
X-Mailer: Internet Mail Service (5.5.2448.0)

Here is a draft 1 spec of the catalog intrastructure. The spec provides
background, motivation, goals, and detailed design coverage. Notably
missing but on its way this weekend is the reference describing the
components, interfaces, methods, and parameters in idl. Also missing are
much needed pictures which will definitely appear repeatedly on whiteboards
and hopefully in code before someone demands we put them in the spec.

Much has happened in the month since the original catalog infrastructure
strawman: an email of terse interface descriptions and wirings. Over a few
meetings the admin team has come to a good understanding of the
infrastructure. A couple more meetings gave the server events team the
understanding to take advantage of it as well. Jan Gray and I also
discussed runtime usage before his departure. The admin team understands all
the components of the infrastructure and their wiring; a descent schedule is
possible. Meanwhile, Cheng Zhou implemented a data table atop the NT DS and
is working on a second-version using marshallable rowsets as the in-memory
cache. This data table will be the basis of the others. Steve Swartz plans
to begin coding next week on a data table using pure oledb to access SQL
Server for server events. I plan to begin coding the dispenser and
marshaller once I complete the Wolfpack support and simplified package
activation for MTS2 Beta 3.

Thanks,
Robert Craig.

MTS3 CATALOG INFRASTRUCTURE SPECIFICATION:

INTRODUCTION: The existing catalog infrastructure:

The Microsoft Transaction Server "catalog" stores and manages the tables of data used by the MTS runtime and admin. The MTS catalog is a client-server architecture which itself takes advantage of MTS for deployment, process isolation, remote activation, and middle-tier logic. Clients can read and write catalog tables and perform high-level operations on the catalog. The catalog is the data service for managing a distributed MTS system.

How is catalog currently implemented? In MTS1 and 2, each MTS computer contains its own single independent catalog. A client can interact with many distributed catalogs and a catalog can service many clients. The catalog schema is presented hierarchically. An administrator manages a set of MTS computers. Each computer contains a list of packages installed, remote components it knows about, and transactions status and statistics. Each package contains a list of components and roles; each component contains a list of interfaces and role memberships; and each interface contains a list of methods and role memberships. A list is a table, its items the rows, the item properties the columns.

Catalog tables are (predominantly) stored in the system registry on each computer. Each table is implemented as a C++ "provider" class derived from a common base. A "provider" of a given table supports a single canned query; reading and writing the table; navigating, inserting, and deleting rows; and getting and setting column values as strings.

Access to the registry itself is via registry nodes, a COM-based registry abstraction layer. A registry node is a cursor into the registry hierarchy which can navigate upward, downward, and sideways and can operate recursively on whole trees at any level. The base "provider" drives a pair of these registry nodes via a declarative "meta" array. If the derived provider information is stored as children under a single registry tree, the base provider coupled with a new meta array is sufficient for most operations. If the provider is assembling information from multiple registry trees, additional code and registry nodes are necessary, but meta arrays can be used to drive column gets and sets. Middle logic, including verifying client changes to column values, triggering updates to other tables, and other custom code is added to the derived class.

The combination of registry nodes, base provider, and meta arrays compose our data tier; the derived providers compose the middle tier. Because the data tier only partially implements the data tier services, the middle tier must implement the remainder, so the distinction of two layers is blurred. And since the middle tier relies on registry nodes too, the middle tier logic is tightly bound to the registry.

Clients need an efficient means of interacting with the catalog tables on any given computer. The "catalog server", a COM component hosted in MTS, meets that need. The catalog server marshalls a table at a time between the client and itself. To read a table the client submits the values of a canned query and gets the resulting table back in a single roundtrip. To write a table the client submits the entire table and gets detailed results of the update back, again in one roundtrip. The catalog server also exposes other interfaces and methods for high-level operations on a catalog. These high-level operations make common but complicated scenarios simple from the client perspective and efficient in roundtrips.

Tables are marshalled as safe arrays of bstrs. We had hoped to use marshallable rowsets, but the technology was not available at the time. The catalog server builds the outgoing table-for-marshalling, aka the safe array of bstrs, from the appropriate provider one column at a time. In turn an incoming marshalled table is packed back into the appropriate provider one column at a time. The catalog server locks the entire catalog on a client request, allowing multiple readers or a single writer. Transactions are not supported.

On the client-side, C++ "accessors" are used to interact with the marshalled catalog tables. The accessor can be thought of as a different flavor of "provider". They sit atop safe arrays of bstrs instead of the registry, passing those arrays back and forth to the appropriate catalog server. Providers and accessors share the same simple interface for table, row, and column interaction. Accessors though are almost entirely driven by meta arrays. The client-side meta information additionally includes presentation-specific meta.

Before the client uses a new catalog server instance, the client first negotiates its session. The session endures until the client releases the catalog server. The client submits the lowest and highest session versions it supports and its lcid (not currently used) and the catalog server returns the version for this session (or that no common version was found). This supports forward and backward compatibility or gracefully determining incompatibility: critical in a distributed environment. A guaranteed set of tables and guaranteed format of those tables is associated with a session version. Because versioning is handled at a high level, versioning issues are absent in lower levels like tables and columns.

INTRODUCTION: Pros and cons of the existing infrastructure:

Much was learned about the existing catalog infrastructure of the course of shipping two major MTS versions. Many MTS features rely upon and were successfully implemented and shipped with the existing infrastructure. Over this time the weakness of that infrastructure have also become apparent.

Pros of the existing catalog infrastructure include:
* The "interface" (albeit C++ class based) for tables is very simple and at the same time comprehensive for our needs.
* Middle-tier logic was absolutely necessary for component and security management; a pure data approach would not work.
* The 3-tier approach (though two tiers were blurred) made implementing fully distributed administration possible.
* Hosting the catalog server in MTS ate our own dogfood and took advantage of MTS simplifying features.
* Marshalling at the granularity of tables yielded acceptable distributed performance.
* Session negotiation for forward/backward/non compatibility versioned all tables and their formats as a unit per session.

Cons of the existing catalog infrastructure include:
* The schema is expressed within the source code itself as a mix of declarations and code (rather than in a datastore).
* Any change to the schema, the internal registry storage structure, the queries, or the middle logic requires new code and rebuilding the product.
* The data and middle tiers are blurred together, and the whole is tightly bound to the registry.
* Converting tables into and out of their marshalling format a column at a time is inefficient.
* Transactions are not supported and adding such support to the existing infrastructure would be very difficult.

INTRODUCTION: Goals for the revised infrastructure:

The general direction of MTS3 is to scale the administration and runtime to support more servers, more clients, and more applications. Another direction for administration specifically is creating a COM Explorer which merges COM, DCOM, and MTS information. Considering this general direction coupled with the pros and cons of the existing catalog infrastructure has led to the following goals in revising that infrastructure:

Support changes of datastore, schema, and logic to an installed system with minimal impact:

* Eliminate the need to recompile the entire infrastructure on any change.
* Support multiple datastores transparently.
* Store schema in the datastores themselves.
* Allow schema to not be stored on client computers.
* Support plug-and-play changing of datastores and middle logic on a per table basis.

Keep the design simple (the delivery timeframe is very short):
* Continue using a very simple interface for table interaction at all tiers.
* Make it very easy to write the code to plug new datastores into the infrastructure.
* Abstract the middle and client tiers from datastore-dependecies.
* Make it much easier to quickly write middle tier logic.
* Continue versioning at the session level to keep lower levels free of it.
* Provide an easy migration path for the existing catalog code base.

Take appopriate advantage of OLEDB and MTS:
* Use oledb for datastore-specific code whenever available and feasible.
* Take advantage of rowsets for caching.
* Use marshallable rowsets for table marshalling.
* Use the MTS runtime as the norm, taking full advantage of its features.
* Continue taking advantage of the n-tier approach facilitated by MTS
* Insure the design supports adding transaction support under-the-covers.

Insure good runtime and admin performance:
* Support the distinctly different performance requirements of the runtime and admin.
* Cache tables in memory; go to the disk for queries and batch updates only.
* Appropriately trade-off minimal network roundtrips for marhalling data with minimal memory for large tables.

DESIGN: Table access via simple tables:

MTS1 and 2 used a very simple but sufficient interface for working with tables. We formalize this for MTS3 into a COM interface, ISimpleTable, with clearly defined semantics. All catalog tables expose this interface, including data, middle, and client tier tables.

Assuming you have obtained one of these tables via a query (detailed below), ISimpleTable provides 9 methods for interactions:
* 2 table-level methods: PopulateTable for executing/re-executing the query and UpdateTable for writing all changes to the datastore. Conceptually, these methods read the table from the datastore to a cache and write changes from the cache to the datastore. All other methods conceptually work with the cached table.
* 3 row-level navigation methods: RestartRow and GetNextRow for forward cursoring and MoveToRow for lookup by primary key.
* 2 row-level manipulation methods: InsertRow and DeleteRow by primary key.
* 2 column-level methods: SetColumn and GetColumn for setting and getting column data respectively by ordinal.

ISimpleTable is at least an order of magnitude simpler than the oledb rowset collection of interfaces and about half as simple as the ADO 1.5 recordset. And while ISimpleTable can be a thin layer atop various flavors of oledb rowset implementations, ISimpleTable layers above non-rowset implementations as well. Because ISimpleTable closely matches our existing provider/accessor model, we have an easy and shippable first step for migrating the sizeable MTS2 catalog code base. Rather than splitting the currently blurred middle and data tiers, we can just slap ISimpleTable on

top and accomplish the split gradually.

DESIGN: Hiding datastores under data tables:

Client and middle tables always sit atop data tables. These upper tables always interact with their data tables through ISimpleTable. Thus only the data tier tables themselves are datastore dependent and specific. The client and middle tables are unaware of the particular datastore on which they operate. And because connecting client and middle tables to data tables occurs outside the tables themselves (detailed below), the underlying datastore can be changed at or after MTS install time on a table-by-table basis.

To plug a new datastore into the catalog infrastructure you implement a new data table exposing ISimpleTable. The schema for tables from the new datastore must be appropriately persisted (usually in the datastore itself), not hardcoded in the data table implementation. This requirement means that only one data table implementation is required per datastore. The data table can therefore read and write any table from the datastore without prior knowledge of it. New tables can be defined in the datastore and subsequently used in the infrastructure without modifying the data table itself.

MTS3 anticipates the need to interact with the following datastores: the registry (for COM information), the component library (for MTS component and package properties), the NT directory service (for locator services), the class store (for code downloads), and SQL server (for server events). Each data table implementation will use oledb to the extent that it is available. The registry lacks oledb completely. The component library supports basic oledb, but does not support commands or sql queries, and only supports indexes on one-column-per-table. The NT DS supports oledb for reading only, but supports ldap (not sql) queries, and lacks indexes for fast lookups; ADSI is required for writing. The class store, though built atop NT DS, requires using its own interfaces. SQL server, sitting under the oledb ODBC provider, is completely oledb.

The data table for SQL server is really a data table for pure oledb. As each datastore migrates toward full oledb support, the custom data table built for that datastore will be replaced with the data table for pure oledb. It is important to note that the catalog infrastructure is not exposed to customers and is not intended to be another data access technology. One purpose of the infrastructure is to allow the majority of our catalog code to access the datastores we require in a simple, common, datastore independent manner, whether the datastore supports complete oledb, partial oledb, or no oledb. Another purpose is to make it very easy to write the necessary code to plug a new datastore into the infrastructure regardless of its level of oledb support.

DESIGN: The different flavors of tables:

You need to obtain a table before taking advantage of its ISimpleTable interface. So how are tables obtained? Regardless of the table, its location in a distributed environment, its datastore, or the desired query, ISimpleTable is always obtained from a single source: the simple table dispenser. The dispenser knows about four flavors of tables: data, middle, client, and top. As an example of table "flavors", consider a table of components, stored in a component library.

The "data table" is the implementation which knows how to interact with tables stored in component libraries. Given the appropriate identifier, locator, and querying information, the data table component can lookup the datastore, schema, and table itself. The data table provides raw access to the table. One data table component exists for each datastore supported by the infrastructure.

The "middle table" is the additional middle tier logic for a given table. Middle tables run on the server-side of the n-tier architecture. They

implement additional code for enforcing rules, do work relevant work beyond the scope of the table, and so on. For example, changing data in the component library table of components may require changes to the registry or interaction with the class store. Not all tables need middle tier logic. One middle table component exists for each table or set of tables requiring special logic. Middle tables can be nested atop each other, and one-to-many mappings are allowed.

The "client table" is the implementation which knows how to read and write tables to and from the server-side of the n-tier architecture. The client table knows how to connect the server-side regardless of its location. This component serves as the front-end of tables marshalled back and forth between client and server. While one client table component would exist for each different flavor of server, only one catalog server is anticipated for MTS3. The catalog server will typically run as an MTS process, support session negotiation, and exchanges tables are marshalled rowsets. The client table is essentially a particular type of data client; one whose datastore is the inherently distributed catalog server itself.

The "top table" is the additional client-side logic for a given table. This is the client-side equivalent to the middle table on the server. Again, not all tables require additional client-side logic. For each one that does, one top table component implements that logic.

DESIGN: Dataspaces and versioning:

A dataspace defines a specific collection of tables and a contract on that collection. Tables may be added to the dataspace, but once added cannot be removed. Columns can be appended onto existing tables, but once added cannot be removed, re-ordered, or re-typed. This contract allows middle and top logic to be written against a guaranteed schema. The contract allows backward and forward compatibility support in a distributed environment with various client and server versions, while restricting versioning knowledge to a small proportion of code. Determining which dataspace to use between a particular client and server occurs as part of the initial negotiation of the catalog session between them. Each major session version has its own dataspace.

While the contract restricts "external" flexibility it allows "internal" flexibility. The datastore a particular table resides in can be changed dynamically at or after install time. And the middle and top logic applied to a particular table can be changed dynamically as well.

Before using a simple table dispenser, the dispenser must be initialized to particular dataspace. ISimpleTableDispenser exposes 5 methods. The first, Initialize, attaches the dispenser to a particular dataspace. Dataspaces are identified by guid. Given a dataspace id, the dispenser can locate all the information necessary to supply any simple table flavor of any table in the dataspace.

DESIGN: The basics of acquiring flavors of a table:

The other 4 methods of ISimpleTableDispenser allow acquiring each of the four flavors of any table in the dataspace. GetDataTable and GetMiddleTable obtain the data and middle flavors respectively; both share the same parameters. GetClientTable and GetTopTable obtain the client and top flavors respectively; both share the same parameters. Tables are identified by guid and all four methods take a table id. All four take consume a query string and query string type (described below). All four methods give back an ISimpleTable pointer ready to use on the requested table.

GetClientTable and GetTopTable differ from GetDataTable and GetMiddleTable in that the former also take a locator string. For MTS3 this locator string is the computer name where the catalog server of interest resides. For a catalog server on the same machine as the client, the locator string is empty.

The simple table dispenser, given a dataspace id, can access the persisted information necessary to supply each flavor of a table. This information can be conceived of as a table itself with a row for each table and the table id as the primary key. The remaining columns include the top, middle, and data table clsids and a locator string.

On a GetDataTable request, the dispenser creates the data table via the clsid, and through ISimpleDataTable::Initialize passes in the table id, query string and type, and locator string. The locator string contains whatever information the implementation requires to get into its database and access the table requested. Note that when a table is moved into a different datastore, the associated data table clsid locator string must be updated in the dataspace.

On a GetMiddleTable request, the dispenser creates and initializes associated data table first. The dispenser then creates the middle table via its clsid. Finally, it calls ISimpleUpperTable::Initialize and submits the table id and query string and type. But instead of a locator string it submits the ISimpleTable pointer of the data table. This is how (in the default scenario) a middle table is connected to its data table.

A GetClientTable request is just like a GetDataTable request, except that the dispenser uses the locator string from the call (which is currently the computer name on which the server resides) instead of from its dataspace information. The client table, being a special data table, is also initialized via ISimpleDataTable::Initialize.

A GetTopTable request is very similar to a GetMiddleTable request. As with middle tables, ISimpleUpperTable::Initialize is used for initialization, except that the ISimpleTable pointer supplied is the client table.

DESIGN: Advanced layering and use of the table flavors:

Neither the top nor middle flavor need exist for a particular table. And a one-to-one mapping from top to client to middle to data table is not necessary. On a request to GetMiddleTable, where no middle table clsid is stored in the dataspace, the dispenser simply returns the data table. The same is true for a top table: GetTopTable returns the client table when no top table clsid is stored.

Data tables can also be lacking. On a request to GetMiddleTable, where no data table clsid is stored, the dispenser calls ISimpleUpperTable::Initialize with NULL instead of the ISimpleTable pointer to a data table. This allows the middle table to sit instead atop one or more middle tables or even multiple data tables. Such a middle table must be smart enough to call the dispenser during its initialize to make the appropriate requests for those tables. This allows arbitrary composition and extension of tables.

Consumers of tables within a client typically always call ISimpleTableDispenser::GetTopTable; they typically should not care whether they get a generic client table or top table logic atop the client table. In a server, ISimpleTableDispenser::GetMiddleTable is typical. This way the caller always gets the middle logic, but only when it exists.

DESIGN: Querying simple tables:

All flavors of ISimpleTableDispenser::Get*Table on the dispenser take a query string and query type. These two parameters are passed down to ISimple*Table::Initialize on the simple table component. Normally only data tables are concerned with query strings; the other flavors just ignore them or pass them down. A query type defines a specific query string format. A data table component first verifies that it supports the query type. If so it expects the associated format when it reads the query string itself.

Many query formats are possible. The NT DS supports ldap; SQL Server supports the be-all-end-all of query formats; other datastores don't support

a query format at all. A least-common-denominator query format among all data tables in the catalog infrastructure is required to truly hide datastore dependencies. The job of a data table implementation is convert this common denominator format into the format appropriate to the data store. SQL syntax, though the most powerful, is too much work to support above every datastore. So while the catalog infrastructure supports any query format, we define a least-common-denominator format which all data tables must support.

The query type, eQT_EQUAND, allows and'ed series of column value equality comparisons. Given a table of interfaces, and the need for only those interfaces on component x in package y in server group z, the query string would (basically) be "servergroup=z & package=y & component=z". A query for interfaces in the package would (basically) be "servergroup=z & package=y". A query for all interfaces at the highest scope visible would be "". The difference between these examples and actual query strings is that the columns are identified by ordinal rather than name. Thus the eQT_EQUAND syntax is "[COLa=VALa[&COLb=VALb[...]]]" where COLx is 0..n and VALb is a string value.

Each data table must retrieve the appropriate tabular subset given an eQT_EQUAND query string. However a data table can limit what specific queries it supports. For example, for hierarchical stores, not all possible queries may be feasible. If the particular query is supported, the data table must do what it takes to carry it out. For the component library, this means taking advantage of the indexed column and then sub-searching the resultset. For the NT DS it means translating it to ldap, and for SQL Server it means translating it to sql.

Note that initializing the data table does not actually execute the query. The query is not executed against the datastore until ISimpleTable::PopulateTable is called. If PopulateTable is called, then changes are made, and PopulateTable is called again, the changes are discarded and the query re-executed. Each PopulateTable call can potentially yield a different resultset, depending on the datastore isolation level.

While a data table instance can requery its datastore repeatedly, changing the query string itself is not supported. Only one query is supported per data table instance.

MTS3 will not support SQL query capabilities like row selection, column filtering and ordering, and table joining in a general manner. MTS3 however will be ready to migrate toward these capabilities as our datastores unify behind oledb and sql and a common query processor.

DESIGN: Marshalling tables between the client and server:

Recall that a client table is just another data table, except that its datastores are a server components distributed thoughout an enterprise with which it exchanges tables. Only one client table will be implemented in MTS3: one which exchanges marshallable rowsets with the catalog server within a session.

The locator string passed to a client table via ISimpleDataTable::Initialize is a computer name in MTS3. The client table attempts to connect to a catalog server on the specified computer, and establishes a session via ICatalogSession. On ISimpleTable::PopulateTable the client table calls ICatalogTableRead and passes along the table id and query string and type and gets back the resulting rowset along with a errors rowset. On ISimpleTable::UpdateTable when the table has changed, the client table calls ICatalogTableWrite and passes along its table id and changed rowset and gets back an errors rowset.

Interestingly, dataspace information need not reside on a client machine. The only dataspace information necessary to obtain client tables from servers on other machines is a default client table clsid. The dataspace

information and schema can both be stored on the server. This fact allows for a much smaller footprint on an administration-only machine. Likewise, the client need not have the dataspace information and schema for all the dataspaces with which it can communicate: again a footprint savings for clients which support several catalog session versions.

The component which implements ICatalogSession/ICatalogTableRead/ICatalogTableWrite is known as the marshaller. On a read request the marshaller uses its simple table dispenser to request a middle table, passing along the table id and query string and type, and calls ISimpleTable::Populate on the table. If the simple table exposes ISimpleTableMarshallableRowset, it then calls ISimpleTableMarshallableRowset::Supply to quickly grab the table data in recordset format and pass it along to the client. This is a convenient way to both easy and speed the marshalling process for data tables which already use a rowset as their table cache. If ISimpleTableMarshallableRowset is not exposed, the marshaller instead walks each row and column of the table a populates a marshallable rowset it creates and defines.

A similar process occurs when a client passes a rowset to the marshaller for writing. The marshaller uses its dispenser to obtain the the appropriate middle table. If the middle table exposes ISimpleTableMarshallableRowset it calls ISimpleTableMarshallableRowset::Consume and passes in the rowset. Otherwise it walks the rowset and submits the changes one-at-a-time via ISimpleTable. Finally, in both cases it calls ISimpleTable::UpdateTable.

DESIGN: Asking a table to describe itself:

Schema/meta interface: TBD.

REFERENCE: Catalog infrastructure components and interfaces (and their MTS2 counterparts):
REFERENCE: ISimpleTable:
REFERENCE: ISimpleTableDispenser:
REFERENCE: ISimpleUpperTableInitialize:
REFERENCE: ISimpleDataTableInitialize:
REFERENCE: ISimpleTableMarshallableRowset:
REFERENCE: ICatalogTableRead:
REFERENCE: ICatalogTableWrite:
REFERENCE: ICatalogSession:

SUMMARY: Putting it all together with an end-to-end example:

TBD.

Todo:
- DESIGN: How can the marshaller get errors when using ISimpleTableMarshallableRowset.
- REFERENCE: Initialize does not cause a Populate.
- REFERENCE: MoveToRow followed by GetNext is not supported.
- SUMMARY: ISimpleTableDispenser::GetClientTable (idST_COMPS, "\\rcraig1", "pak=pakid", eST_QTYPE_CAT_EQUAND, &pIST).
- ADD PICTURES.

COM+ Catalog Infrastructure (Robert Craig, draft 4)

INTRODUCTION TO USING THE CATALOG INFRASTRUCTURE:

The COM+ Catalog, evolved from the MTS Catalog, is the distributed multi-tiered database for COM+ configuration data. All of the magic which manipulates, manages, and serves up this configuation data sits atop a common "catalog infrastructure". Special requirements govern the design of this catalog infrastructure. A code sample introduces these requirements.

The following code sample asks the COM+ Catalog for a table of all library applications installed in our three favorite server groups. (Library applications run in the caller's process; a server group consists of a group of computers all running a common set of applications and governed by a common configuration.) First we obtain the table, then we simply iterate over its rows, and for each row iterate over its column values. The code below is complete except for error checking unanticipated conditions:

```
ISimpleTableDispenser*          pISimpleTableDispenser = NULL;
ISimpleTable*                   pISimpleTable;
ULONG                           i, j;
LPVOID                          pvData;
WCHAR                           wszQuery [128];
HRESULT                         hr;

hr = CoCreateInstance (
        CLSID_SimpleTableDispenser, NULL, CLSCTX_INPROC_SERVER,
        IID_ISimpleTableDispenser, (void**) &pISimpleTableDispenser);

wsprintf (wszQuery, L"%d=CorpExchange|RCraigA|ComTestBed & %d=%d",
        iCOL_DSNAME, iCOL_ACTIVATION, eACTIVATION_LIBRARY);
hr = pISimpleTableDispenser->GetClientTable (
        didSERVERGROUP, tidAPPLICATIONS,
        wszQuery, eQF_SIMPLE_ANDOR, fST_READONLY, &pISimpleTable);
if (E_ST_DBNOTSUPPORTED == hr) return;

for (i = 0; ;i++)
{
        hr = pSimpleTable->GetNextRow ();
        if (E_ST_NOMORE == hr) break;
        wprintf (L"ROW %d:", i);
        for (j = 0; j < cCOLUMNS_APPLICATIONS; j++)
        {
                hr = pISimpleTable->GetColumn (j, NULL, NULL, &pvData);
                wprintf (L"\tColumn %d:\t%s\n", j, (LPCWSTR) pvData);
        }
}
pISimpleTable->Release ();
pISimpleTableDispenser->Release ();
```

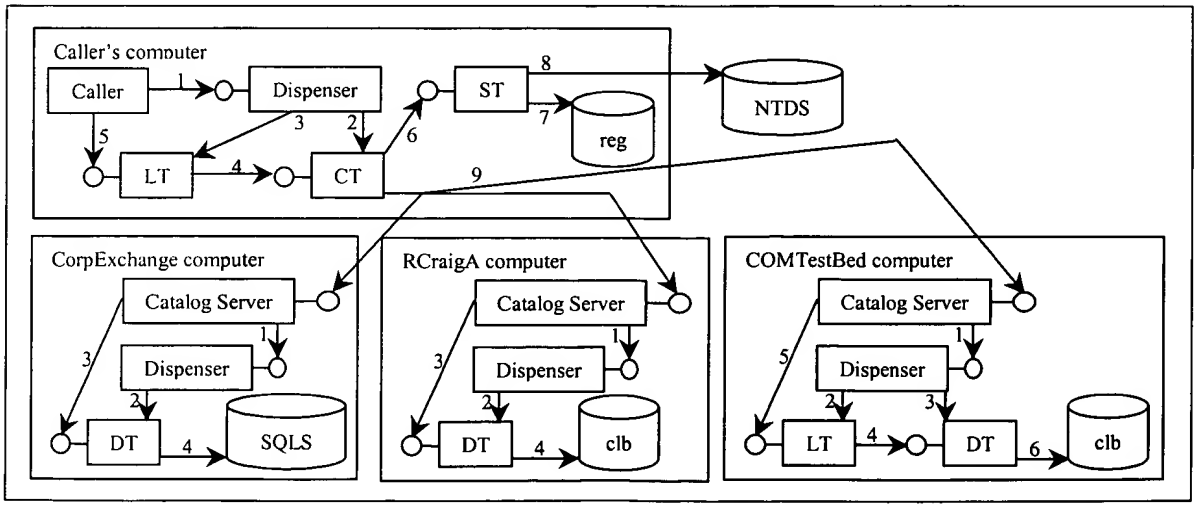
Let's briefly review the code. First we create a simple table dispenser, the means for obtaining all catalog tables. Since we are interested in assembling a set of rows distributed among 3 server groups, we request a "client" table from the dispenser. Client tables allow gathering data from multiple network locations. We specify a database id (DID), which identifies a guaranteed minimum collection of well-defined tables. We specify a table id (TID), which identifies a minimum collection of well-defined columns. We use a simplified query syntax to specify the directory service names of the 3 server groups and request only library applications. Finally, we request a read-only table, since we are just printing the contents. We receive an ISimpleTable pointer, a cursor to a new cache of rows retrieved by our query. We call GetNextRow to iterate over all rows. For each row we call GetColumn to iterate over all column values. We could use a meta-information interface to determine the number of columns, but because we know this TID, we know its number of columns. GetColumn also allows us to obtain the type and size of a column value, but in this case we also know all values are strings. We print the column values, which we received by reference, not copy. Finally, we clean everything up.

INTRODUCTION TO HOW THE CATALOG INFRASTRUCTURE WORKS:
The catalog infrastructure revolves around the concept of tables. The sample code above illustrates first obtaining then using a table through this infrastructure. While the code to obtain our table is minimal, obviously a lot happens under-the-covers to execute this distributed query. Two of the "infrastructure" components involved are apparent in the code: 1) *Simple tables*: a simple and uniform cached-based means of working with tables from different datastores and locations; 2) *Simple table dispensers*: The means of obtaining simple tables. Hidden beneath the code is the fact that multiple simple tables, dispensers, and datastores participate in this query. These include *data tables*, *logic tables*, and *catalog servers*.

Simple tables come in two distinct flavors: "data tables" and "logic tables". Data tables, internally, are datastore dependent. A particular data table works with a particular datastore. Externally though, the code for working with any of these data tables is the same. In fact, the caller never knows from what datastore a particular table originates. The dispenser has the job of binding a datastore-independent request by a caller to the appropriate datastore-dependent data table, and providing enough information to that data table so it can bind to its appropriate datastore. The client table illustrated in the code is just another data table. Client tables work with a special kind of datastore called "catalog servers". Residing one per computer, the catalog server provides one-stop-shopping for COM+ catalog tables residing on that computer. The client table can communicate remotely with catalog servers, exchanging marshalled tables back and forth.

Logic tables do what their name implies. They sit atop tables, enforcing logic specific to their table. While a data table works with a particular datastore, a logic table works with a particular table, like the applications table in the sample code. Logic tables can be plugged in on both the client and server sides (ie: in the caller's process and in the catalog server), and they can be layered. A logic table can intercept any caller action on a simple table, enforcing domain-specific rules and processing. And while under the covers they alter behavior, the code for working with logic tables is the same. Again, the caller never knows how many if any logic tables exist atop the data table they requested. The dispenser has the job of wiring in these logic tables using wiring information from its own database. From the caller perspective, its table is always just a simple table.

So what happens under the covers in the code sample? First the dispenser creates a client table and wires in any client-side logic. On population, the client table needs to locate the computers on which the master databases for the 3 server groups in question reside. It does so by looking first in its cache and then querying the NT DS, via simple tables. It then calls the catalog server on each master database computer. Each catalog server passes the query on to its dispenser to get the table. The dispenser associates the DID+TID to a data table, creates and binds the data table to its datastore, wires in any server-side logic tables, and populates the table against the query. The catalog server marshalls the results back to the client table. The client table combines the results into a single table ready for cached-oriented action from the caller. Note that dispensers access their wiring database via simple tables. Also, each server group may store its applications table in a different datastore, and each may have different server-side logic. Briefly illustrated:



REQUIREMENTS:

The catalog infrastructure revolves around the concept of obtaining and working with tables. Using the sample code above as illustration, the basic requirements of the catalog infrastructure are:

1. Lightweight, easy to use, easy to implement: Tables must be usable via a small number of interfaces and methods. Data should be cached at the point of use, and access to that cache should be direct. The sample code illustrates this by using 7 methods and 7 variables to execute a distributed query and iterate over all resulting values. Finally, implementing the infrastructure must be easy as well, meaning a small number of interfaces and methods and simple, clearly defined semantics. While the job must be easy, the result must be powerful: eg: queryable, marshallable, cache-oriented, batched updates, fast single and multi-cursor read performance, and so on.
2. Datastore independent and transparent (dynamically): The catalog must be able to take advantage of the right datastore for each job. No datastore provides COM+ with a one-size-fits-all solution, and datastores are a shifting landscape. COM and MTS currently use typelibs and the registry for configuration data: typelibs describe today's COM components; the registry runs on every computer and makes prototyping easy. COM+ will make extensive use of component libraries (clbs): their trademark being incredible read performance and a small footprint. Enterprises will want to use SQL Server as the master datastore describing their distributed applications: SQL Server provides reliability and scalability. Locating distributed resources should take advantage of the NT Directory Service. Application-wide vs computer-wide vs enterprise-wide databases all have different requirements which result in distinctly different implementations.

   Given this shifting landscape, datastore independence is critical. As little code as possible should be tightly coupled to a particular datastore. Coupling to a datastore should be dynamic rather than permanent. And coupling should be configurable on a per-table basis. All of this leads to datastore transparency: middle and client tiers are unaware from which datastore a table originates, allowing the origin to differ on different machines or even on the same machine at different times. The sample code demonstrates this datastore transparency.
3. Datastore location independence and transparency: COM+ configuration data is not centralized. By necessity, some resides on clients and a lot resides on servers. For example, COM+ server groups also require a combination of centralization and distribution. A server group is a set of applications and a set of server computers which run those applications. Managing a server group occurs through a master configuration database. However each server has its own persistently cached subset of the configuration data required to run its applications.

   Given this distributed landscape, eliminating location dependencies is critical. Manipulating data from another computer must have reasonable performance. Requesting data from different computers should be just as easy as requesting data locally. And while some requests are location specific, many requests simply need the data regardless of its location. Datastore location should be transparent in such requests. In the sample code, remote calls occur only in GetClientTable: iterating over the data uses an in-process cache. GetClientTable executes the query requesting data from 3 server groups. Finding and accessing the appropriate datastores is the responsibility of the catalog infrastructure.
4. Plug-and-playable middle- and client-tier logic: The COM+ catalog contains by necessity very rich domain-specific middle and client tier logic acting on its tables. For example, modifying an application may trigger changes to the security system, cause file download or transfer, etc. Middle and client tier logic on tables should be pluggable as much as possible, as opposed to hardcoded. This allows for the transparent dynamic addition of logic over time, including post-ship. The catalog infrastructure is responsible for wiring this logic. So in the sample code, while retrieval from two of the server groups may be from their datastores directly, the third might pass through a middle-tier filtering layer, transparent to the caller.
5. Scalable and reliable infrastructure: The COM+ catalog, as a distributed multi-tiered database, obviously must be scalable and reliable. While not illustrated in the sample code, distributed notifications, security, and transactions all help fulfill this requirement. The catalog infrastructure expects most of this capability to be provided a combination of its underlying datastores and COM+ plumbing. When these capabilities must be exposed through the infrastructure, an easy to use and common model must be supplied. The infrastructure may also need to fill in gaps where these capabilities are missing from particular underlying technologies.

FEATURES:

The following features of the catalog infrastructure help meet the abovementioned COM+ Catalog requirements:

{WIP:

1. Invariant table identity including well-defined schema.
2. Uniform, multi-cursor, cached-based access to tables.
3. Table, row, column, and cursor operations exposed via one interface.
4. Lightweight concurrent access model; simple concurrent change control.
5. Few column types, with conversion support to and from Unicode strings.
6. C++ oriented table access to pure, partial, and non-OLEDB datastores.
7. Cached accumulation of changes for batched datastore updates.
8. Access to column values by copy or direct reference.

9. No versioning concerns from most table-related code.
10. Under-the-covers binding of table requests to datastores.
11. Catalog management centralized in catalog servers.
12. Remote table access to tables in any catalog server.
13. Support for multiple query formats.
14. Simple-yet-powerful catalog-wide query format.

15. Datastore-specific code tightly centralized, not spread everywhere.
16. Relatively small amount of code to integrate a particular datastore.
17. Datastore-specific tables and connections transparently poolable.
18. Location information centrally maintained yet datastore specific.

19. <All the data table section stuff goes here.>

20. Client and middle tier logic can be datastore independent.

}

{WIP:

21. Plug-and-play of datastore-independent middle- and client-tier logic.
22. Efficient remote access to tables via marshalled caches.
23. An easy to use, implemented-once distributed query format.
24. Comprehensive per-table-instance meta information.
25. Propogation of aggregate error information on populate and update.

26. Notification to interested parties of datastore updates on a per-table basis.
27. Security, including row-level.
28. Automatic distributed transactions.
29. Tables derived from joins (tbd).
30. Within-cache row filtering and sorting (tbd).
31. An api layer for high-level multi-table catalog operations.
32. An api layer answering a fixed set of performance-critical runtime requests.
33. Exposure of the COM+ Catalog as an OLEDB provider.
34. Take maximum advantage of existing and emerging OLEDB implementations.
35. Support plug-and-play generic table services (hard to do).
36. Catalog meta tables for defining schema, setting up datastores, and discovering schema (tbd).
37. Dogfooding: Use of other COM+ services such as distributed automatic tx, RET, and LCE.

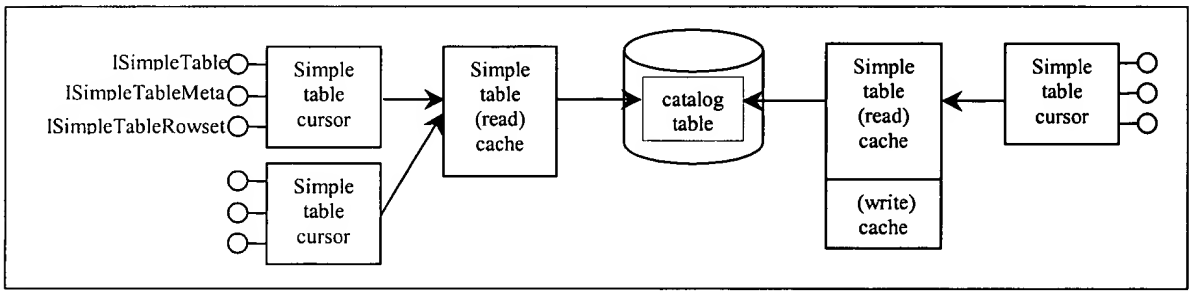
}

HOW TO WORK WITH A TABLE:

Table identity: All catalog tables are identified by table id (TID), a globally-unique id. A given TID guarantees a minimum set of well-defined columns. The order of these columns (beginning at 0), their types (DBTYPEs) and type-related meta-information are all immutable. This means that even when requests for the same TID are made to different datastores, different computers, and even different COM+ versions, the resulting table will always be the same. Well, almost. New columns can be added to an existing table, provided: 1. they are ordered after the existing set of columns; 2. existing clients expecting only the original set of columns do not break; and 3. they become part of the new set of well-defined columns. If the well-defined set of columns otherwise changes, that table can no longer be identified by the same TID.

Row identity: Rows in a given TID must each be uniquely identifiable by some fixed subset of their column data. One or more specific columns must contribute to row identity. At most one row in a table has a particular combination of data in these columns. These column values can never be NULL. Again, if this well-defined set of identity columns changes, that table can no longer be identified by the same TID.

Caches and cursors: Working with a catalog table (ignoring for a moment how tables are obtained) occurs via a "simple table". A simple table is an in-memory cached snapshot of a table, rather than the table-in-the-datastore itself. All access to this cache occurs via a "simple table cursor". A simple table cursor is a non-COM-createable COM component (ie: not creatable via CoCreateInstance, but accessible via COM interfaces). A simple table cursor exposes three interfaces: ISimpleTable, ISimpleTableMeta, and ISimpleTableRowset (the latter two interfaces are described elsewhere in this document). Multiple cursors are supported on the same cache, and multiple caches are supported on the same table in a given datastore. Briefly illustrated:



The main interface: ISimpleTable provides all the functionality a typical caller requires to work with a table, its rows, and its columns. The interface consists of 12 methods:

- 3 table-level methods: *PopulateCache* reads the table, constricted by query, from the datastore into the cache; *UpdateStore* writes changes made in the cache to the datastore. No other methods interact with the datastore. *UpdateReadCache* updates the read cache with changes in the write cache.
- 5 row-level navigation methods: *RestartRow* positions the cursor at the beginning of the table; *GetNextRow* moves the cursor forward to the next row; *MoveToRowByIdentity* moves the cursor to a row by its row identity; *MoveToRowByIndex* moves the cursor to a row by its position; *MoveToNewRow* moves the cursor to a new row.
- 2 row-level "change" methods: *SetRow* commits all changes made to the current row to the cache (not the datastore); *DeleteRow* deletes the current row.
- 2 column-level methods: *GetColumn* and *SetColumn* get and set column data on the current row.
- 1 cursor-level method: *CloneCursor* supplies another cursor to the same read-only cache.

Marshalling: ISimpleTable is strictly an in-process interface. From the caller perspective, their ISimpleTable pointer points to an in-process cursor to an in-memory cache. *PopulateCache* and *UpdateStore*, in exchanging data between the cache and datastore, may go out-of-process under-the-covers. Callers can anticipate that these two calls may be expensive and that all other calls will be cheap. (The default "under-the-covers" cross-machine "marshalling" strategy is covered elsewhere in this document.)

Read and write caches: When a caller obtains a table they specify either a read-only or read-write cache. A read-write cache actually consists internally of two caches: a read and a write cache. The read cache only changes on *PopulateCache* (which obtains a cached snapshot of a table by (re-)executing the query against its datastore) and *UpdateReadCache* (which updates the read cache with the changes in the write cache). The write cache changes on *SetRow* (which either inserts a new row or changes an existing row) and *DeleteRow* (which marks a row as deleted). The write cache also changes on *PopulateCache* and *UpdateStore*, both of which discard all changes in the write cache (the latter after writing them to the datastore). The write cache does not change on *MoveToNewRow* and *SetColumn*, both of which only change the cursor (the subsequent call to *SetRow* transfers those changes into the write cache). The write cache is not readable via ISimpleTable and its changes do not appear in the read cache except on an *UpdateReadCache* or a *PopulateCache*.

Types and conversions: All data tables must support the following subset of OLEDB data types: DBTYPE_UI4, DBTYPE_WSTR, DBTYPE_GUID, DBTYPE_BYTES, and DBTYPE_DBTIMESTAMP. (This subset derives from support requirements and may change as critical requirements arise.) If a data table's underlying datastore does not support these types, the data table is responsible for under-the-covers conversion to and from these types when populating the cache and updating the store. Simple tables do not support type conversion when getting and setting data.

Getting and setting data: *GetColumn* always supplies a 4-byte value. If the column's type always requires only 4 bytes, *GetColumn* supplies the data by value. Otherwise, *GetColumn* supplies the data by read-only reference (ie: a 4-byte pointer to the data). This reference only remains valid during the time the caller holds the cursor and during that time only until *PopulateCache* or *UpdateReadCache* are called by any cursor on that cache. The caller must make their own copy if they are to reference the data beyond that window of time. The caller must never write to the referenced data.
SetColumn always consumes a 4-byte value. Again, if the column's type always requires only 4 bytes, *SetColumn* assumes the 4-bytes consumed are the data itself. Otherwise, *SetColumn* assumes the 4-bytes provided are a reference to the data. The cursor holds these references until *SetRow* is called, at which time it copies the data and discards the references. The caller must therefore retain all such data passed by reference to *SetColumn* until after calling *SetRow*.
NULL pointers are accepted for and to be expected of all references. This pass-by-reference design obviously reduces allocate-and-copy numbers to a minimum.

Concurrency control: A simple table cursor is a stateful component useful only to one thread at a time. However a simple table cache is useable by multiple cursors (each created via *CloneCursor*) and therefore by multiple threads. Recall that *GetColumn* provides direct references to the read cache. These references are valid only during the lifetime of the cursor and only until *PopulateCache* or *UpdateReadCache* is called by any cursor on that cache. Also recall that since *SetRow*, *DeleteRow*, *UpdateStore*, and *PopulateCache* all change the write cache, these calls must be serialized among all cursors on the cache.
Responsibility for all concurrency control is left to the multi-threaded caller. The caller must manage its entire set of cursors according to the following rules. 1) A cursor must be owned by only one thread at a time. 2) All cursors but one must be blocked during a *PopulateCache* or *UpdateReadCache*, and all those cursors must subsequently immediately call *RestartRow*. 3) A call to *SetRow*, *DeleteRow*, *UpdateStore*, or *PopulateCache* must block any other call to any of those methods. 4) All references to cache data must be treated as read-only and must be forgotten before any call to *PopulateCache*, *UpdateStore*, or *UpdateReadCache*, or that cursor's call to *Release*.
Concurrency control is only necessary for multi-threaded read-write usage scenarios and multi-threaded multi-*PopulateCache* read-only scenarios. Since these scenarios require some caller-based concurrency control anyway, this approach eliminates double locking in these scenarios and eliminates all locking overhead from other usage scenarios.

Cooperative vs competing changes: The same row in a cache can be repeatedly changed before an update of the datastore from the cache occurs. A cache deals with multiple changes to the same row using "last writer wins". This means that each change to the same row writes over the last change. However, a datastore deals with multiple changes to the same row using "optimistic concurrency". This means that if an update from a cache attempts to change a row which was already changed after the cache's population

but before its update, the change is rejected. The cache vs datastore strategies differ because multiple changes to the former are assumed to be cooperative, whereas multiple changes to the latter are assumed to be competitive.

Wrapping various OLEDB providers: The simplicity and specificity of ISimpleTable and its semantics benefit both the caller and implementor. ISimpleTable is far simpler than the OLEDB rowset collection of interfaces (although it layers well above them) and similar in simplicity to ADO but more specific and oriented toward optimized C++ programming. Simple tables can be layered atop non-OLEDB datastores, atop datastores which expose an insufficient subset of OLEDB for the required semantics, and atop datastores which expose sufficient OLEDB for all required semantics. In the final scenario of a pure OLEDB datastore, ISimpleTable merely wraps the work necessary to use the general OLEDB interfaces in the very specific manner common throughout the COM+ catalog.

ISimpleTable: The interface specification follows:

```
HRESULT PopulateCache ();
/* Populates the read cache from the datastore, using the database, table, and query
specified to the dispenser.  The previous cache contents, including pending changes, are
discarded.  Restarts the row cursor.  This is the only ISimpleTable method which reads
from the datastore.  The dispenser makes the first (and typically only) call to
PopulateCache before supplying the ISimpleTable cursor to its caller.
*/
HRESULT UpdateStore ();
/* Writes all pending changes in the write cache to the datastore and forgets those
changes. Returns E_NOTIMPL on a read-only cache.  This is the only ISimpleTable method
which writes to the datastore.
*/
HRESULT UpdateReadCache ();
/* Updates the read cache with the pending changes in the write cache.  Restarts the row
cursor.  Note that neither the datastore nor write cache change.  Returns E_NOTIMPL on a
read-only cache.
*/
HRESULT RestartRow ();
/* Restart the row cursor just prior to the first row in the read cache.
*/
HRESULT GetNextRow ();
/* Moves the row cursor from the current row to next row in the read cache.  When
preceeded by RestartRow, moves the cursor to the first row.  When the cursor is on the
last row or a new row, GetNextRow does not move the cursor and returns E_ST_NOMORE.
*/
HRESULT MoveToRowByIdentity (ULONG* i_acb, LPVOID* i_apv);
/* Moves the row cursor to the unique row in the cache matching the identity specified.
The parameter requirements for this method are described elsewhere in this document.  If
the row does not exist, returns E_ST_NOMORE and does not move the cursor.
*/
HRESULT MoveToRowByIndex (ULONG i_iRow);
/* Moves the row cursor to the row i_iRow's from the first row in the cache.  If i_iRow
is not less than the number of rows, returns E_ST_NOMORE and does not move the cursor.
*/
HRESULT MoveToNewRow ();
/* Moves the cursor to a new row held by the cursor.  SetRow inserts the row into the
write cache.  All non-default columns must first be set with SetColumn.  SetRow must be
called before the cursor moves.  Returns E_NOTIMPL on a read-only cache.
*/
HRESULT DeleteRow ();
/* Marks current row deleted in the write cache.  Returns E_NOTIMPL on a read-only cache.
*/
HRESULT SetRow ();
/* Updates the write cache (not the datastore) with the changes made to the current row.
If the row is from MoveToNewRow, but the row is already identified in the read cache, the
write cache is still updated with changes to the row.  Returns E_NOTIMPL on a read-only
cache.
*/
HRESULT SetColumn (ULONG i_iColumn, ULONG i_cb, LPVOID i_pv);
/* Prepares to set column i_iColumn of the current row using the supplied data.  The
change is held by the cursor until SetRow is called.  If the data type always only
requires 4 bytes, the caller passes the data directly; otherwise they pass the data by
reference, and must retain the data until SetRow is called.  i_cb is only specified for
```

```
non-self-size-describing types (ie: DBTYPE_BYTES) and should otherwise be 0.  If
i_iColumn is out of range, returns E_ST_NOMORE.  Returns E_NOTIMPL on a read-only cache.
*/
HRESULT GetColumn (ULONG i_iColumn, DWORD* o_pdbtype, ULONG* o_pcb , LPVOID* o_ppv);
/* Gets the data for column i_iColumn of the current row. o_pdbtype receives the type,
and may be NULL; o_pcb receives the size of the data itself and may be NULL; *o_ppv
receives the data if its type always only requires 4 bytes, and otherwise receives a
reference to the data, possibly NULL.  If i_iColumn is out of range, returns E_ST_NOMORE.
*/
HRESULT CloneCursor (ISimpleTable** o_ppIST);
/* Supplies another cursor, initially at the same location in the read cache as the
current cursor.
*/
```

HOW TO OBTAIN A TABLE:

Databases vs datastores: A globally unique database id (DID) identifies each catalog database. A given DID guarantees a minimum well-defined set of catalog tables, each table being identified by and complying to the rules for a table id (TID). A DID therefore represents a well-defined and invariant database schema. Removing any table from such a database violates its identity. A DID also guarantees at least one query format (detailed below). A DID is a datastore-independent identity, meaning that the tables of that database can be distributed among multiple datastores. Examples of datastores include: the registry, type libraries, SQL Server, and the NT Directory Service, whereas examples of databases include: server group databases, download databases, and deployment units.

Versioning: Because a database id guarantees a minimum set of tables and their complete schema, once a caller identifies a database they are assured all the tables they expect will be available. This allows versioning concerns to be restricted to a relatively small amount of code. Database ids also enable backward and forward compatibility in a distributed environment, where differing client and server versions of applications co-exist.

Data tables: The simple table dispenser binds a client's request for a datastore-independent database and table to the appropriate datastore-dependent "data table" and datastore, according to wiring instructions in that dispenser's database. A data table is a simple table implementation which knows how to work with a specific datastore. Each type of datastore has its own simple table implementation (eg: SQL Server, NT Directory Service, component libraries, registry). Each of these implementations expose the simple table cursor and exactly supports the simple table semantics. Callers can therefore interact identically with all data tables. The simple table cursor's interfaces completely hide which data table, and therefore which datastore, the caller is using.

Catalog servers: Residing one-per-computer, a catalog server owns and manages all COM+ databases on its computer. All administration requests, whether local or from another computer, go to the catalog server. The catalog server also serves as a special datastore: one through which all tables in all COM+ databases on its computer can be accessed, both locally and remotely. A special data table, called a "client" table, knows how to work with any catalog server on any computer, serving its tables up as simple tables.

Client vs server tables: Callers can get tables via two methods on the dispenser: ISimpleTableDispenser:: *GetServerTable* and *GetClientTable*. Both methods share the same parameters. Both methods consume a database id, table id, and query, and both supply a simple table cursor to a new cache populated with the requested table. They differ as follows. *GetServerTable* supplies, on the current computer, the appropriate data table bound to the appropriate datastore. *GetClientTable* supplies a client table bound to the appropriate local or remote catalog server(s), which in turn binds to the appropriate "server" table. Server tables are subject to server-side and not client-side logic, whereas client tables are subject to both.

Simple table consumers: Who uses client and server tables? The catalog server itself uses server tables. The catalog runtime API, which supplies catalog data to performance critical COM+ runtime consumers, also uses server tables. The COM+ Explorer, the COM+ Administration SDK, and the COM+ Catalog OLEDB provider use client tables. The rest of the world accesses the COM+ Catalog through one of these higher layers.

Queries: GetClientTable and GetServerTable accepts a variety of query formats. Callers specify a unicode query string and a unique identifier of its format. While this model allows for any query format, including SQL (the most powerful query language on the planet), the catalog infrastructure defines a simplified query format, called "simple and/or", guaranteed to be supported catalog-wide. The "simple and/or" query format, while relatively powerful, is relatively easy to implement against all anticipated COM+ catalog datastores. Once enough OLEDB infrastructure exists, SQL will replace this query format as the catalog-wide standard.

"Simple and/or" queries: eQF_SIMPLE_ANDOR identifies the "simple and/or" format. Its syntax can be expressed as: "[COLa=VALa1[|VALa2[..]][&COLb=VALb1[|VALb2[..]]]]", where COLx is a column ordinal and VALxn is the string form of a column value. For example, given a table of component

information, we might want to query for all components in applications a, b, and c which require a transaction and whose threading model is free or both: "3=a|b|c & 9=required & 8=free|both" where column ordinals 3, 9, and 8 are named by constants iCOL_APPNAME, iCOL_TX, and iCOL_THREADMODEL respectively. The syntax is case insensitive. DBTYPE_UI4 values appear as integers; DBTYPE_GUID values must include curly braces; DBTYPE_DBTIMESTAMP has the "hh:mm:ss mm/dd/yy" format; DBTYPE_BYTES appears as hex.

Reserved queries: Certain reserved column ordinals exist which are not part of any table but are used to locate the appropriate table. iCOL_CMPNAME and iCOL_DSNAME only appear in client table queries. iCOL_CMPNAME specifies a computer name. iCOL_DSNAME specifies a directory service name which resolves (the client table knows how to do the resolution) to a iCOL_CMPNAME. iCOL_FILE specifies a file name. iCOL_INDEX specifies an index hint whose value is a column name. Data tables which do not support index hints simply ignore this column name. Both a NULL query string and empty query string populate the cache with all rows in the table.

Table flags: GetClientTable and GetServerTable also accept table flags. Currently supported flags include:

- **fST_TABLE_READONLY**: Obtains a read-only table (default is read-write). This allows overhead reduction in read-only scenarios, such as for performance critical runtimes. The following methods on ISimpleTable and ISimpleTableMeta return E_NOTIMPL when this flag is specified: *UpdateStore*, *UpdateReadCache*, *MoveToNewRow*, *DeleteRow*, *SetRow*, *SetColumn*, *ForgetPendings*, *RestartPendingRow*, *GetNextPendingRow*, and *GetPendingColumn*.
- **fST_TABLE_NONMARSHALLABLE**: Obtains a non-marshallable table (default is read-write). This allows overhead reduction in non-marshalling scenarios, such as for performance critical runtimes. *ConsumeMarshallable* and *SupplyMarshallable* on ISimpleTableRowset return E_NOTIMPL when this flag is specified.
- **fST_TABLE_SINGLECURSOR**: Obtains a single cursor table (default is multi cursor). This allows overhead reduction in single cursor scenarios. ISimpleTable::*CloneCursor* returns E_NOTIMPL.
- **fST_TABLE_EMPTY**: Obtains an empty cache ready for writes (default is a populated cache). This allows overhead reduction in non-reading writer scenarios. The dispenser still calls *PopulateCache*: the data table however must ignore the query and supply an empty cache ready for writes.
- **fST_TABLE_INCOMING**: Means the first client call to the simple table will be ISimpleTableRowset::*ConsumeMarshallable*. This advanced option, allowing for overhead reduction when the catalog server updates tables from clients, should normally never be used. To specify this flag and not call *ConsumeMarshallable* is a programming error.
- **fST_TABLE_NOLOGIC**: obtains a data table directly sans client and/or server tier logic. This advanced option, intended to allow short-circuiting shipped logic via isolated patches, should normally never be used.

ISimpleTableDispenser: The interface specification follows:

```
HRESULT GetServerTable (
        REFGUID i_did,
        REFGUID i_tid,
        LPCWSTR i_wszQuery,
        DWORD i_eQueryFormat,
        DWORD i_fTable,
        ISimpleTable** o_ppIST);
/* Gets a populated server table, used by the catalog server itself or the catalog
runtime API.  i_did is the database id; i_tid is the table id; i_wszQuery and
i_eQueryFormat are the query string and format; i_fTable is table flags; o_ppIST points
to an ISimpleTable pointer on success.  Returns E_ST_DBNOTSUPPORTED if the database or
table specified is not recognized.
*/
HRESULT GetClientTable (...);
/* Same as GetServerTable, except gets a populated client table, used by the COM+
Explorer, COM+ Admin SDK, and COM+ catalog OLEDB provider.  Accepts queries to remote
catalog servers.  Goes to the local catalog server by default.
*/
```

HOW DATA TABLES WORK:

Implementing a data table: Plugging a particular type of datastore into the catalog infrastructure requires implementing a data table specific to that type of datastore. Such an implementation typically includes three pieces: the cache, the simple table cursor, and the data table dispenser.

Populating and updating: Of all the ISimpleTable methods on a data table, *PopulateCache* and *UpdateStore* are the only methods which actually interact with the datastore itself. All the other methods interact with the cache and its cursors. (In fact, an appendix of this document describes how to derive from a base data table which implements all cache and cursor functionality, leaving only the job of *PopulateCache* and *UpdateStore* to the deriver.) Both *PopulateCache* and *UpdateStore* need enough information to locate the appropriate datastore, table, schema, and subset of rows. The simple table dispenser provides the start of this information when it initializes the data table.

Obtaining a data table: While callers work with data tables via ISimpleTable cursors, these cursors are not COM createable components themselves. Callers always obtain the first cursor to a particular cache via the simple table dispenser. The dispenser obtains this cursor via a COM createable "data table dispenser" which exposes ISimpleDataTableDispenser. Each data table implementor provides its own data table dispenser. The simple table dispenser creates this data table dispenser, queries for ISimpleDataTableDispenser, calls *GetTable* with the information necessary to obtain the appropriate simple table cursor to a new cache, and finally calls the cursor's *PopulateCache* to execute the query.

Locator information: In the call to ISimpleDataTableDispenser::*GetTable*, the dispenser passes in the database id, table id, and query string and format from the caller along with special locator information . The data table implementor uses the locator information to bind to the appropriate datastore, schema, and table. The implementor defines their locator information format and content. The locator can be as granular as table specific; however some implementors may only require datastore specific locators, and for some implementors, the table id and query from the caller may be sufficient. The implementor defines these locators. The dispenser simply retrieves these locators from its wiring database and calls *GetTable* with the locator associated with the caller-requested database, table, and query.

Implementing a data table dispenser: Each time a caller requests a table, the simple table dispenser creates the appropriate data table dispenser, calls ISimpleDataTableDispenser::*GetTable*, and then releases the data table dispenser. This sequence allows for a variety of under-the-cover possibilities. In the simplest case, the data table dispenser and the dispensed data table itself can actually be implemented on the same component. However, the dispenser might also be a singleton, or a pool of per-datastore singletons. The dispenser might also pool its caches, perhaps on a per-TID basis, and/or its connections, typically on a per-datastore basis.

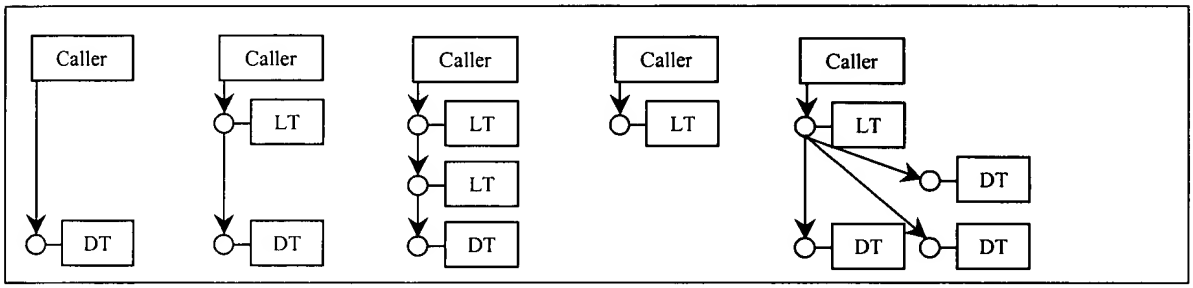
ISimpleDataTableDispenser: The interface specification follows:

```
HRESULT GetTable (
        REFGUID         i_did,
        REFGUID         i_tid,
        LPCWSTR         i_wszQuery,
        DWORD           i_eQueryFormat,
        DWORD           i_fTable,
        LPCWSTR         i_wszLocator,
        ISimpleTable** o_ppIST);
/* Gets a simple table cursor to a new cache bound to the appropriate datastore and ready
for population from and/or updates to the requested database table.  i_did and i_tid are
database and table ids respectively; i_wszQuery and i_eQueryFormat are the query and
query format; i_fTable specifies data table flags (fST_TABLE _READONLY, _NONMARSHALLABLE,
_EMTPY, _INCOMING); i_wszLocator is the implementation-specific locator information.
o_ppIST points to an ISimpleTable cursor on success.  Returns E_ST_DBNOTSUPPORTED if the
database or table specified is not recognized.
If the requested table has not been created in the datastore, population must still
succeed with an empty cache, and updates must result in datastore table creation.
*/
```

HOW LOGIC TABLES WORK:

Domain-specific logic: Domain-specific rules and processing can be transparently added to a particular table or set of tables. Such domain-specific "logic" typically exceeds the capabilities and scope of the underlying data tables. Examples include: complex relationships among column values in a row may need enforced when callers change column values; server-side row or column filtering may be necessary depending on who is reading a table; complex relationships among tables may require special enforcement and management as those tables change; certain table changes may trigger other changes outside the scope of the catalog's tables. Transparent incorporation of domain-specific logic occurs via "logic tables".

Using a logic table: When a caller asks the simple table dispenser for a table, it either receives a cursor to a data table or a logic table. The caller cannot tell which, just as it cannot tell to which datastore a particular table is bound. So while the caller's code looks the same, the under-the-covers picture can vary dramatically. The caller could be connected to a data table directly, to a logic table wired atop the data table, to a series of logic tables all wired one atop the other, or even to a logic table wired atop nothing or wired atop multiple tables. Logic tables may be wired into both client and server tables for client-side vs server-side rules and processing. Logic tables may also be excluded from read-only tables, since most domain-specific processing occurs in write scenarios. The dispenser supports these different wiring pictures under-the-covers (the details of this wiring are covered elsewhere in this document). Briefly illustrated, possible under-the-cover wiring pictures for logic tables include:



Implementing a logic table: While data table implementations are particular to a datastore, logic table implementations are typically particular to a specific table or set of tables. And while a data table implementor typically builds three pieces (the cache, the simple table cursor, and the data table dispenser), the logic table implementor typically builds only two: the simple table cursor and a logic table dispenser. A cache is not applicable, since the logic table typically sits atop one or more data tables, merely introducing domain-specific rules and processes to the underlying cache. The logic table's cursor relies on the underlying data table cursor for access to the cache. The logic table's cursor merely allows the implementor to intercept all calls intended for the underlying data table's cursor, so that they may introduce logic and delegation as they see fit.

Obtaining a logic table: As with data tables, callers work with a logic table via a simple table cursor. Recall that simple table cursors are not COM createable and that callers always rely on the simple table dispenser for the first cursor to a particular cache. When one or more logic tables exist for a given table id in a given scenario, the dispenser, after obtaining the appropriate data table via its data table dispenser, wires these logic tables atop that data table. An implementor of a set of logic tables must write a COM createable logic table dispenser. The simple table dispenser creates this logic table dispenser, queries for ISimpleLogicTableDispenser, calls *GetTable* to obtain the desired logic table, then releases the dispenser.

Implementing a logic table dispenser: ISimpleLogicTableDispenser::*GetTable* receives the database and table ids, query string, and query format from the caller. Passing through the database id, table id, and query allows one logic table dispenser to supply multiple logic tables and allows a logic table implementation to service multiple tables. *GetTable* also receives the ISimpleTable cursor to the table atop which the logic table will sit. The logic table dispenser responds to *GetTable* by supplying an ISimpleTable cursor of the appropriate logic table implementation, layered atop the received ISimpleTable cursor. In the simplest wiring scenario, the caller ends up with the logic table's cursor instead of the data table's cursor.

Above and beneath the logic table: The only access a logic table has to its underlying table is via an ISimpleTable cursor. The logic table therefore cannnot tell to which datastore it is bound. The logic table might sit directly above its data table, or it may sit atop another logic table who further intercepts its downward calls. Also, the logic table might sit directly beneath the caller, or it may sit beneath another logic table who delegates calls downward to it. The typical logic table has no knowledge of who exactly is above or beneath itself.

Interception and delegation: A logic table must implement all the simple table cursor interfaces: ISimpleTable, ISimpleTableMeta, and ISimpleTableRowset. This means the implementor intercepts all cursor method calls by default. Most calls the implementor simply directly delegates to the simple table cursor underneath with which they were initialized. On a few calls they actually do work, passing on the call before or after or not at all, possibly altering parameters when they do so. The most popular methods for interception include ISimpleTable::*UpdateStore*, *SetRow*, and *PopulateCache*. The logic table must also always intercept all IUnknown methods and ISimpleTable::*CloneCursor* to prevent being removed from interception when the caller changes interfaces or creates new cursors.

Optimized delegation: Logic tables use a delegation rather than aggregation approach. Delegation centralizes the layering code in the overlaid table, whereas aggregation spreads it to the underlying table as well. Also, aggregation only optimizes calls to interfaces the overlaid table does not intercept at all. The easiest direct delegation approach is to just call the underlying method from the overlaid method. However this incurs the cost of an extra indirection per layer. This indirection on direct delegation can be eliminated by fixing up the overlaid table's vtable entry to reference the underlying table's method directly. An appendix of this document describes how to derive from a base class which implements this fixup capability. The delegation approach with direct delegation fixups is optimal in that only those methods desired for interception incur any delegation cost and this optimization is hidden from both the simple table dispenser and the underlying table.

ISimpleLogicTableDispenser: The interface specification follows:

```
HRESULT GetTable (
        REFGUID         i_did,
        REFGUID         i_tid,
        LPCWSTR         i_wszQuery,
        DWORD           i_eQueryFormat,
        DWORD           i_fTable,
        ISimpleTable*   i_pIST,
        ISimpleTable**  o_ppIST);
/* Consumes an ISimpleTable cursor and supplies a new replacement ISimpleTable cursor
which introduces domain specific logic.  i_did and i_tid are database and table ids
respectively; i_wszQuery and i_eQueryFormat are the query and query format; i_fTable
specifies logic table flags (none currently); i_pIST is the underlying ISimpleTable
cursor.  o_ppIST points to an ISimpleTable cursor on success.
*/
```

HOW THE SIMPLE TABLE DISPENSER WORKS:

Dynamic table wiring: The simple table dispenser has the job of wiring a caller's datastore-independent request for a table to the appropriate datastore-dependent data table and then wiring in the logic appropriate to that table. Obviously the dispenser requires a healthy amount wiring information to accomplish its job. This wiring information resides in a database, allowing dynamic configuration. Since each computer where catalog databases reside has its own wiring database, the recognized databases and the underlying stores can vary from computer to computer. The dispenser reads its wiring tables using read-only simple tables (dogfooding). Because the wiring tables are accessible via simple tables, configuration changes can also be made using simple tables. The dispenser itself is implemented as a singleton: at most one instance runs in a given process. This minimizes memory consumption since the dispenser, for efficient access, caches its wiring database information as necessary.

Computer-wide vs file-wide databases: The catalog infrastructure supports two different classes of databases: computer-wide and file-wide. Recall a database is a guaranteed minimum set of tables, not to be confused with a datastore. For a computer-wide database, at most one instance of that database can reside on a particular computer. The COM+ registration database is an example. For a file-wide database, at most one instance of that database can reside within a particular file. The COM+ deployment unit is an example.

For a given database instance, a set of wirings exist which allows the simple table dispenser to wire requests for any table in that database to the appropriate datastore. For a computer-wide database, only one set of wirings exists on a given computer. These wirings differ across computers when different datastores are chosen and when different versions of the database instance exists. For a file-wide database, multiple sets of wirings can exist on given computer: one for each version of the file. For example, the deployment unit for COM+ 98 will contain both tlbs and clbs, whereas in COM+ 99 it will likely contain only clbs. For file-wide databases, the dispenser needs some extra help to determine which set of wirings is appropriate for a given file.

Client-side vs server-side dispensers: A single simple table dispenser implementation exists which serves up both client and server tables. However the necessary wiring information differs dramatically between client and server tables. The dispenser's wiring database is fractured along these lines. For a client computer which lacks certain databases itself but accesses them via other catalog servers, this fracturing means only the client-side wiring need reside on that client computer, reducing disk footprint and installation complexity. For a client or server process on a computer, this fracturing means the dispenser in that process only sees the appropriate information, reducing memory footprint. Besides the wiring information itself, the behavior of the dispenser in serving up client vs server tables also differs.

Finding database wirings: The simple table dispenser requires some bootstrapping instructions. These instructions reside in a well-known, fixed registry location. This is the only persistent information the dispenser does not obtain via simple tables. A data table dispenser and locator are specified for the dispenser's "database wiring" table. This "database wiring" table contains the general wiring instructions for each database and is the starting point for accessing the per-table wiring instructions in any given database. A default client data table dispenser and locator are also specified. For COM+ 98, the same "client table" will be used for all tables: namely the implementation which knows how to communicate with catalog servers. Since the dispenser is not hard-coded to default to that implementation, the default client table can be replaced on a per-computer basis without replacing the dispenser itself. Conceptually, the bootstrapping schema of this single-row table is:

The "database wiring bootstrap" table schema (tidSTD_DBWIRING_BOOTSTRAP; rowid 0):

0	G	n	The data table dispenser for the database wiring table.
1	S	n	The locator for the database wiring table.
2	G	n	The default client data table dispenser.
3	S	y	The default client data table locator.

(Note: For schema grids in this document: the columns from left to right represent the column ordinal, the column type (G is DBTYPE_GUID, S is DBTYPE_WSTR, U is DBTYPE_UI4 and so on), whether or not NULL or 0 is allowed (ie: the value is optional), and a brief column description.)

Using database wirings: The general wiring instructions for a given database reside in the "database wiring" table. The remaining wiring instructions, specific to each table, divide among the following tables: basic table wiring for client and server tables (two tables) and extended logic table wiring for client and server tables (two tables). These four tables are optional. Each database instance known to the dispenser has its own row in the "database wiring" table and may also have any of these four tables. The wiring instructions for all databases known to the dispenser conceptually reside in the same "databases wiring" database (didSTD_DBWIRING).

Database wiring identity: The "database wiring" table, derived from the bootstrap instructions, contains the general wiring instructions for all known databases. A database wiring id (DWID) identifies the database wiring instructions for a particular database. For a computer-wide database, the DWID is the same as the DID. For a file-wide database, these ids differ. This allows the same file-wide database to have distinct wiring information on the same computer for each version of the file. Thus a single DID may have several distinct sets of database wiring instructions.
For each DWID, a simple data table dispenser and locator are specified. The dispenser uses this information to access the wiring tables specific to that DWID as simple tables. This allows wiring instructions to reside in different datastores on a per database instance basis. Therefore a database implementor can retain the bulk of their wiring instructions in their datastore of preference. They only need insert their general wiring information in the "database wiring" table to be useable via the simple table dispenser. Note the one caveat: all their wiring tables must be accessible via the same dispenser and locator (the tables must reside together in the same datastore and must be accessible via the same locator and did).

General wiring instructions: In the "database wiring" table, each DWID also has useful general wiring instructions. Recall that wiring tables exist for data and logic tables on client and server sides. A special flag for each such wiring table indicates whether that table is empty or not. These flags let the dispenser know whether or not it should bother getting each of those tables.
What happens when one or more of these wiring tables is devoid of table specific instructions? The general wiring information also includes a default data table dispenser and locator for both client and server tables. Thus, for any database instance, data table wiring can be defined on a per table basis, on a database-wide basis with per table overrides, or simply on a database-wide basis for all tables. For client data table wiring, a database-wide default is unnecessary as the dispenser defines the default client data table wiring in its bootstrapping instructions. In fact, if the database is remote, and therefore no database wiring exists on caller's computer, the dispenser will simply use the default client wiring. For server data table wiring, unless this is defined for every table individually, a database-wide default is required, since no dispenser default exists. Unlike data table wiring, logic table wiring is completely optional.
The general wiring instructions also include flags which, when set, instructs the dispenser to concatenate the default locator for client or server data tables and the table-specific locator. This allows datastores which have lengthy locators to save space and eliminate redundancy in their table wiring instructions.

The "database wiring" table schema (tidSTD_DBWIRING; rowid 0):

0	G	n	The database wiring id (same as DID for computer-wide databases).
1	G	n	The data table dispenser for the database wiring tables.
2	S	n	The locator for these database wiring tables.
3	U	y	Non-zero when the "client data table wiring" table has rows.
4	U	y	Non-zero when the "client logic table wiring" table has rows.
5	U	y	Non-zero when the "server data table wiring" table has rows.
6	U	y	Non-zero when the "server logic table wiring" table has rows.
7	G	y	The default client data table dispenser.
8	G	y	The default server data table dispenser.
9	S	y	The default client data table locator.
10	S	y	The default server data table locator.
11	U	y	Non-zero means concatenate the default client locator with the table-specific locator.
12	U	y	Non-zero means concatenate the default server locator with the table-specific locator.

Basic per-table wiring: Each table in a database can have its own wiring instructions which override the database-wide wiring instructions. Basic per-table wiring instructions are split between two tables having the same schema: the "basic client table wiring" table and "basic server table wiring" table. The TID serves as row identity. For each tid, a data table dispenser and locator can be specified. When not present, the database-wide defaults are used. In the client wiring table, the data table dispenser and locator are an override of the default client table (thus these entries will not be used for COM+ 98). In the server wiring table, these entries are used whenever a database is spread among more than one datastore.
Also included in the basic wiring instruction is an optional logic table dispenser, a flag indicating if extended logic wiring is necessary for this table, and a flag indicating an intention not to use a data table. When no logic wiring exists, the dispenser column is NULL and both flags are false. When one logic table exists, the dispenser is specified, and the extended logic wiring flag is false. When multiple logic tables exist, extended logic wiring is necessary. Indicating a data table is not to be wired up is only valid when at least one logic table exists. Another flag exists which indicates that the logic is also intended to be wired into read-only requests. Most logic tables are only applicable to read-write scenarios. In the client wiring table, the logic wiring obviously specifies client-side logic tables; likewise in server wiring, the logic wiring specifies server-side logic tables.

The "basic table wiring" table schema (tidSTD_BCTWIRING; tidSTD_BSTWIRING; rowid 0):

0	G	n	The table id.
1	G	y	The data table dispenser.
2	S	y	The data table locator.
3	G	y	The logic table dispenser (must be NULL when extended logic wiring is necessary).
4	U	y	Non-zero when logic wiring is also to occur with read-only table requests.
5	U	y	1 when serial logic wiring is necessary.
6	U	y	Non-zero when the intention is not to use a data table (at least one logic table must exist).

Extended logic wiring: Currently only serial logic table wiring is supported. (Hierarchical wiring is not supported as part of the wiring database at this time and must be accomplished within a specialized logic table dispenser). The "serial client logic wiring" table and "serial server logic wiring" table provide wiring instructions for these series on the client and server sides respectively. A row consists of the relevant table id, a logic table dispenser, and the order in which to wire that dispenser. The order begins at 0 and increases, where 0 is directly atop the data table. Row identity in these tables is a combination of TID and ordering.

The "serial logic wiring" table schema (tidSTD_SCLWIRING; tidSTD_SSLWIRING; rowid 0, 1):

0	G	n	The table id.
1	G	n	The logic table dispenser.
2	U	n	The relative ordering atop the data table, from 0, where 0 is directly atop the data table.

File-wide database wiring: File-wide databases require some extra work to determine which wiring instructions to use. This is because the same database can be supported in different files or different versions of the same file using different datastores underneath. The dispenser's "database wiring" table does not include these databases by DID. To support multiple wirings, these databases are included by DWID. The DID specified by the client must be mapped to DWID appropriate to the specified file.

File database wiring detection: To determine whether such a mapping is necessary, the dispenser scans the query string for the iCOL_FILE column. When present, the dispenser knows it does not yet have the DWID and so consults its "database wiring detection" table. Each row specifies a file extension, a database wiring detector, and a detection order. The dispenser considers only those rows matching the file extension. Proceeding through those rows in order, the dispenser creates each detector component indicated in turn. On these components the dispenser calls IDatabaseWiringDetector::*DetectFileDatabase*, submitting the DID and filename. When one of the detectors recognizes the file, it supplies the DWID,

concluding the dispenser's search. If only one row exists for a given file extension, and the detector is not specified, the DID and DWID are assumed by the dispenser to be the same.

The "database wiring detection" table schema (tidSTD_DWIDDETECT; rowid 0, 1):

0	S	n	The file extension.
1	U	n	The detection order, from 0, where 0 is the first detector.
2	G	n	The database wiring detector.

IDatabaseWiringDetector: The interface specification follows:

```
HRESULT DetectFileDatabase (
        REFGUID         i_did,
        LPCWSTR         i_wszFile,
        REFGUID*        o_pdwid);
/* Consumes a database id and file path, and if the file database is recognized, returns
the appropriate database wiring id.  Returns E_FAIL when not recognized.
*/
```

HOW THE CLIENT TABLE AND CATALOG SERVER WORK TOGETHER:

- It is actually the default client table.
- How do we guarantee a remote dataspace with minimal roundtrips?
- Delayed rendering of tables is the responsibility of the Fox rowset.
- Must detail ISimpleTableRowset.
- Must describe implications of fST_NONMARSHALLABLE.

The catalog server and its databases: The catalog server is the server application through which administration of COM+ databases occurs. Each computer containing one or more COM+ databases has a catalog server. The catalog server supports high-level operations on its databases through a set of interfaces. The catalog server also controls reading and writing of the tables in those databases. The catalog server is the means through which remote clients access the COM+ catalog on a given computer. The COM+ catalog itself contains multiple databases. A database is a virtual concept, a collection of related tables, not necessarily all held by the same underlying datastore. Some of these databases may be single instance per computer, such as the COM+ registration database. Others may have multiple instances per computer, such as application deployment units.

HOW TO DISCOVER META INFORMATION ON A TABLE:

- Defaults:
  - On GetColumnMeta, another column meta flag: fST_COLUMN_HASDEFAULT.
  - How to put the default values into the base data table and have it set them?
  - Should we support defaults and sets of valid values at the IST level?
- Primary and foreign keys:
  - We need an fST_COLUMN_FOREIGNKEY and PRIMARYKEY.
  - Allow new and delete on fks and pks but not updates (ie: not writeable)?
  - Include parent and child relationships in meta?
- Does referential integrity need to be part of the exposed schema?
- On GetPendingColumn, another column meta flag: fST_COLUMN_CHANGED vs _UNCHANGED.
- Must clearly note all schema necessary for a TID.
- Concurrency control:
  - No writes while a cursor browses the pendings.
  - No RestartPendingRows while a cursor browses?
- Flag rules:
  - Pass-by-ref pks are always non-nullable.
  - Sized-by-type columns are always fixed length.

```
[local] interface ISimpleTableMeta : IUnknown
{
//      ForgetPendings: Leaves  changes in the cache, but forgets they are pending.
//      Several changes followed by ForgetPendings followed by UpdateTable means no changes
//      are written to the datastore.
        [id(1)]         HRESULT                 ForgetPendings ();


//      RestartPendingRow: Moves the "cursor" prior to the first row with pending changes.
        [id(2)]         HRESULT                 RestartPendingRow ();

//      GetNextPendingRow: Moves the "cursor" to the next pending row.  Returns E_INVALIDARG
//      when no more pending rows exist.  The pending status indicates whether this row was
//      inserted, changed, or deleted.
        [id(3)]         HRESULT                 GetNextPendingRow ( [out] DWORD* o_ppendingstatus);

//      GetPendingColumn: Given a column id and a data buffer with its type and size, put the
//      column data into the buffer.  Returns E_INVALIDARG if the column id is out of range.

        [id(4)]         HRESULT                 GetPendingColumn (
                                                        [in]  ULONG i_iColumn,
                                                        [out] DWORD* o_pdbtype,
                                                        [out] ULONG* o_pcb,
```

```
                                                [out] LPVOID o_pv,
                                                [out] DWORD* o_pFlags);

//      GetRowMeta: Supplies the count of rows and count of columns in this table.  Either
//      parameter may be NULL.  Note that count of rows can be expensive but will always be exact.
        [id(5)]         HRESULT                 GetRowMeta (
                                                [out] ULONG* o_cApproxRows,
                                                [out] ULONG* o_cColumns,
                                                [out] ULONG* o_cUniquenessColumns);

//      GetColumnMeta: Given a column id, supplies the column name, type, maximum size, and a
//      set a flags.  Note that the column name is passed by reference, not copy.  Flags currently
//      include: fST_COLUMN_FIXEDLENGTH (the column data is guaranteed to be a fixed length),
//      fST_COLUMN_PRIMARYKEY (the column data is one of the columns which contributes to the
//      unique data-based identification of the row), fST_COLUMN_NOTNULLABLE (the column value may not be null).
// Other flags will be added as necessary.
        [id(6)]         HRESULT                 GetColumnMeta (
                                                [in] ULONG              i_iColumn,
                                                [out] LPWSTR*           o_pwszName,
                                                [out] DWORD*            o_pdbtype,
                                                [out] ULONG*            o_pcb,
                                                [out] DWORD*            o_pFlags);
// GetTableMeta: Get the table ID, query and query type.
        [id(7)]         HRESULT                 GetTableMeta (
                                                [out]LPWSTR* o_pwszTID,
                                                [out]LPWSTR* o_pwszQuery,
                                                [out]DWORD* o_pQueryType);


};
```

HOW ERROR PROPOGATION WORKS:

- How do we communicate errors on populate and update failures (esp. optimistic concurrency failures)? What about SetRow schema violations?
- How do we pass row and column level error information to the client?
- Should doc all known IST* hr's.
- Use oledb errors.

HOW TO USE NOTIFICATIONS:

- Is table-level not granular enough?
- Notifications are supported for changes to the datastore, not changes to the cache.

HOW TO USE TRANSACTIONS:

- The COM+ admin code can assume automatic transactions are available.

HOW SECURITY WORKS:

FILTERING AND SORTING:

- Sort by single column alphabetically?
- Must describe MoveToRowByIdentity here!

JOINS:

THE SCHEMA DATABASE:

HOW THE CATALOG SUPPORTS COMPLEX ADMINISTRATIONS:

HOW PERFORMANCE-CRITICAL RUNTIMES USE THE CATALOG:

THE ADMINISTRATION SDK:

THE CATALOG OLEDB PROVIDER:

APPENDIX 0: WHY IS THIS SO COMPLICATED?

APPENDIX 1: COM+ CATALOG HISTORY:
Because ISimpleTable closely matches our existing provider/accessor model, we have an easy and shippable first step for migrating the sizeable MTS2 catalog code base. Rather than splitting the currently blurred middle and data tiers, we can just slap ISimpleTable on top and accomplish the split gradually.

APPENDIX 2: OLEDB AND THE CATALOG INFRASTRUCTURE:
- Non-goals...
- Comparison with ADO, OSP, etc.

- Take advantage of oledb wherever possible and reasonable.
- Use the Fox rowset as a table cache and means for marshalling tables.

APPENDIX 3: USING THE BASE DATA TABLE:

Shaping the base simple table: If you are rolling your own DBCOLUMNINFO's for the base simple table to consume, here is what is expected of each field. Only 5 of the 9 fields are used: those fields all have specific requirements.
- *pwszName*: Optional; ignored within simple tables.
- *iOrdinal*: The column ordinal. The bookmark column is always ordinal 0; the remaining column ordinals are contiguous beginning with 1. For simple tables, column ordering is permanent. Column ordinals therefore serve as column ids.
- *dwFlags*: All flags are ignored except DBCOLUMNFLAGS_ISFIXEDLENGTH. Use this flag for all fixed length data types and for DBTYPE_WSTR and DBTYPE_BYTES when all values always have the same length.
- *ulColumnSize*: The maximum possible length of a value in the column. For DBTYPE_WSTR this must either be the maximum length of the column in bytes or ~0 (bitwise 0) when the length is unbounded. Otherwise the oledb conventions for this field apply.
- *wType*: The column's data type: a DBTYPE. For simple tables, the type is permanent. This field never includes DBTYPE_BYREF. Simple tables only support the DBTYPE_WSTR type for strings: DBTYPE_BSTR and DBTYPE_STR are not supported.
- *pTypeInfo*: Ignored.
- *bPrecision*: Ignored.
- *bScale*: Ignored.
- *columnid*: Ignored. The columnid always DBKIND_NAME whose name is pwszName.

APPENDIX 4: USING THE BASE LOGIC TABLE:

APPENDIX 5: SUPPORTED DATASTORES:
Currently needed: TLB, CLB, Registry, NTDS, SQLS, Catsrv.
Currently (partially) implemented: Registry, CLB, NTDS, MTSv2, Catsrv.

APPENDIX 6: USAGE SCENARIOS:
Making data split between the tlb and clb appear as a single table.
Scenarios where you need to go directly to the DT dispenser.
Plenty more...
}

APPENDIX 7: KNOWN PERFORMANCE BOTTLENECKS:
Recognized design-level performance bottlenecks and their issues:
- Getting and setting column values is column-at-a-time. Each get or set of a column requires a function call. The design opts to not reduce function calls to coarser granularity in favor of ease of use and not

exposing internal cache structures to callers. The performance hit is mitigated by the fact that all gets and sets can be optimized to only copy 4 bytes without allocations. The design of passing by reference all values bigger than 4 bytes and of providing only by-reference access to the read and write cache make this possible. Also while column values are set a column at a time, row data which must be copied can be copied a row at a time using SetRow.

- Reading and writing data is row-at-a-time. No means currently exist to bulk read or write a set of rows. This can still be accomplished internally when actually populating from or updating to the datastore. But this optimization is not available to callers.
- Query string parsing. This chews time. Query formats have been expanded to encompass any query structure (including strings). The query structure chosen for COM98 alleviates string parsing. It is a simple boiled down format.
- Population cannot be accumulated over time. No capability for populating a read cache over time using different queries to accumulate more and more matching rows into the same cache is supported. Doing so would have implications on references already held on data in cache. Still investigating a good means for doing this. The alternative is separate table instances for each separate query. Internally these instances could be optimized to share common data and therefore reduce per-table memory overhead to an insignificant amount.
- Asyncronous fetching of rows not supported. The caller is blocked until all rows are fetched. Asyncronous fetching of rows is only necessary when the number of rows is huge and fetching them is costly and the caller needs all rows together. Range fetching is a better solution when not all rows from a huge set are required, which is the typical ui scenario. You never want to fetch a million rows, even asyncronously, when the user will only interact with a few hundred.
- Range fetching not supported. Range fetching would allow the caller to specify the range of rows to fetch. Settling the isolation behavior is the big issue here. However this could certainly be supported via a reserved columnid in the query structure. Accumulating multiple range fetches into an existing cache is nonetheless not supported (see above), so the caller would still need to discard the previous cache and their references to its data.
- Count of rows without populating the table. To get the count of rows in a cache you need to populate the cache, which is inefficient if you only want the count. However counts can be supplied alone simply using a separate table containing only the desired counts.

Performance issues with the fox rowset (as of rds 2.0):

- Provider-owned by-reference access to data is not supported. Even if it were, simple table callers expect by-reference access to the cache itself. A non-trusted-caller rowset design would still require locking, allocating, and copying internally.
- IRowset::InsertRow is rather slow, even when forgetting changes on each row in the load scenario.
- There is a 10K minimum dynamic data overhead just to create an empty fox rowset. Additionally, the code working set is rather large. Several dependent dlls are necessary, some of which are not allowed within the context of some of our consumers.
- The fox rowset maintains its own heap for data. This heap currently grows but never shrinks during the lifetime of the process. For a long-lived process this can be very undesirable, especially when the difference between the average rowset size and largest rowset size is significant.
- Asyncronous fetch over http is available; over dcom is on its way. Range fetches are not supported natively so far as I know.
- I believe the oledb service provider architecture will allow native use of the fox rowset by the database provider based on properties specified. Kagera currently does so via a remoteable property. I am not sure whether the native sqls provider supports this. I am not sure whether this has become a documented part of the service provider architecture and therefore useable by all. Last time I checked this was behavior internal to Kagera.

COM+ 1.0 Catalog

Robert Craig

Development lead

COM+ Catalog Services

Purpose of this Talk

- A "chalk talk" of the COM+ Catalog:
  - What it is.
  - What problems it solves.
  - A little about how it solves them.
  - And time to discuss topics of interest...

What does "Catalog" mean?

- From MTS: A misnomer which stuck…
- American Heritage: "A systemized list"…
- Yet another systems management database?

The COM+ 1.0 catalog is a "virtualized" "database" of COM+ applications and their services, with runtime and configuration-time abstraction layers for using and manipulating this configuration information.

Slide Outline

- Features: Past and Present:

  Talk about catalog features, both internal and customer visible, with a historical perspective, to give an understanding of the problems the catalog solves.

- Design Elements:

  Give the big picture of the catalog architecture followed by an introduction to each of its major design elements, to give an understanding of how the catalog works.

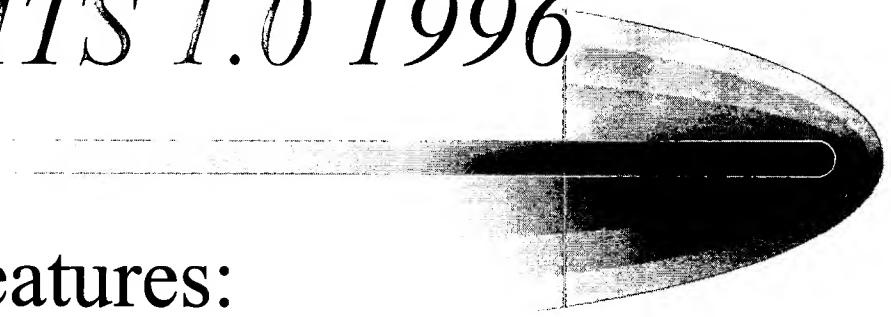
- Key Philosophies and Discussion:

  Summarize the key philosophies that have emerged over the last 3 years of shipping the catalog, to initiate discussion on current and future design topics and issues.

History: MTS 1.0 1996

- External "above-the-catalog" features:
  - MTS 1.0 "services" included declarative transactions and isolation, role-based security, etc.
  - Exposed to customers via MTS admin SDK as opposed to registry hacking.
  - Application deployment via text-based "manifest" and the necessary files grouped together.
  - Remote many-to-many administration support.

History: MTS 1.0 1996



- Internal "catalog" features:
  - Catalog stored in the registry via a new HKLM sub-tree and severe "morphing" of HKCR; also drew on type libraries and the secure section of the registry.
  - Internal, tabular, C++ classed-based, network/relational abstraction model.
  - Reasonable support for joining "tables" and for embedding administration and runtime "logic".
  - Used directly by MTS runtime and "explorer" UI.
  - Table marshalling via automation safe arrays.
  - Isolated and secured via MTS services.

History: MTS 2.0 NTOP 1997

- External "above-the-catalog" features:
  - Supported IIS transactional web applications.
  - MTS services become the basic plumbing for client-server applications (context, clusters, etc).
  - Application proxy deployment via self-extracting self-installing iexpress executables.
- Internal "catalog" features:
  - Performance improvements.
  - Supported more complicated schema / services.

Now: COM+ 1.0 NT5 1998

- "Above-the-catalog" external features:
  - Supporting a much richer schema for advanced server programming services: object construction/pooling, load-balancing, queuing, IMDB, loosely-coupled events, syncronization, etc.
  - Application and application proxy deployment via component library-based "manifest" cabbed with the necessary files into an MSI-based executable.
  - Initial support for self-describing components.
  - Excellent runtime working set and performance.

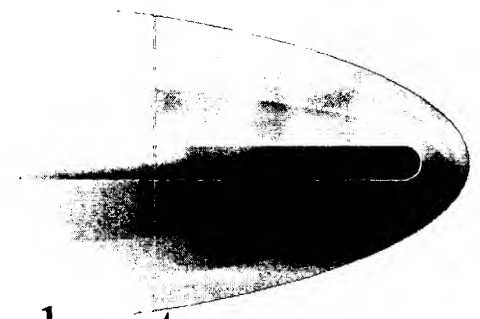
Now: COM+ 1.0 NT5 1998

- Catalog internal features:
  - Hooked directly into COM activation via a "runtime catalog" abstraction layer.
  - Runtime catalog infrastructure does not itself use the COM runtime library itself.
  - Machine-level configuration information stored in regdb: very fast and lightweight system database built atop component library technology.
  - Isolated, secured, *transacted* via COM+ services.
  - Administration UI built entirely atop admin SDK.

Now: COM+ 1.0 NT5 1998

- Catalog "infrastructure" features **(simple tables)**:
  - COM-based relational, query-able, remote-able, tabular abstraction layer.
  - Light-weight, easy to use, easy to implement. Runtime performance considered a top priority.
  - Data-store independent and transparent, with dynamic-binding down to individual columns.
  - Data-store *location* independence and transparency.
  - Plug-and-play client/server administration logic.
  - Scalable and reliable infrastructure by dogfooding COM+ services.

Slide Outline



- Features: Past and Present:

  Talk about catalog features, both internal and customer visible, with a historical perspective, to give an understanding of the problems the catalog solves.

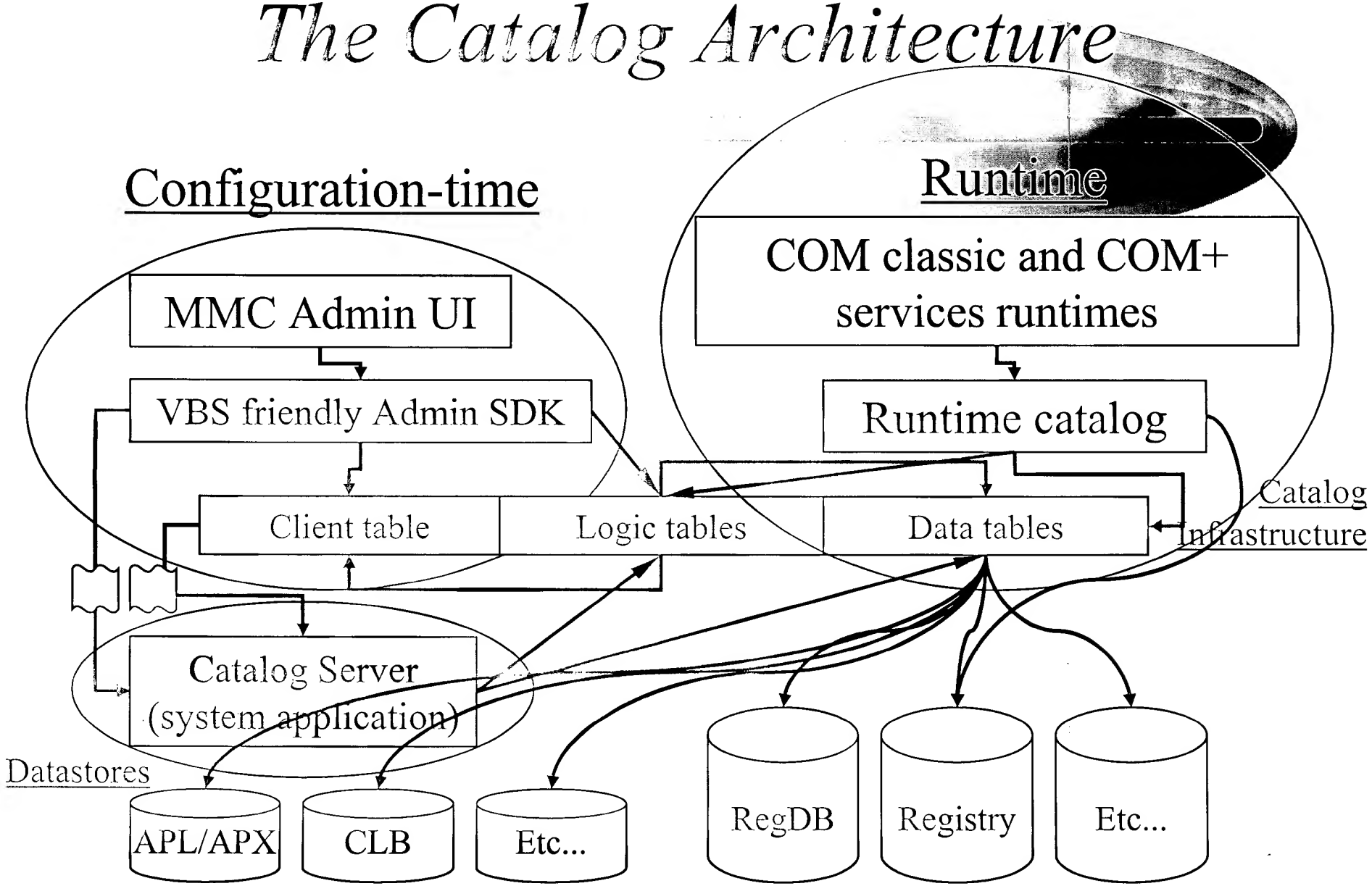
- Design Elements:

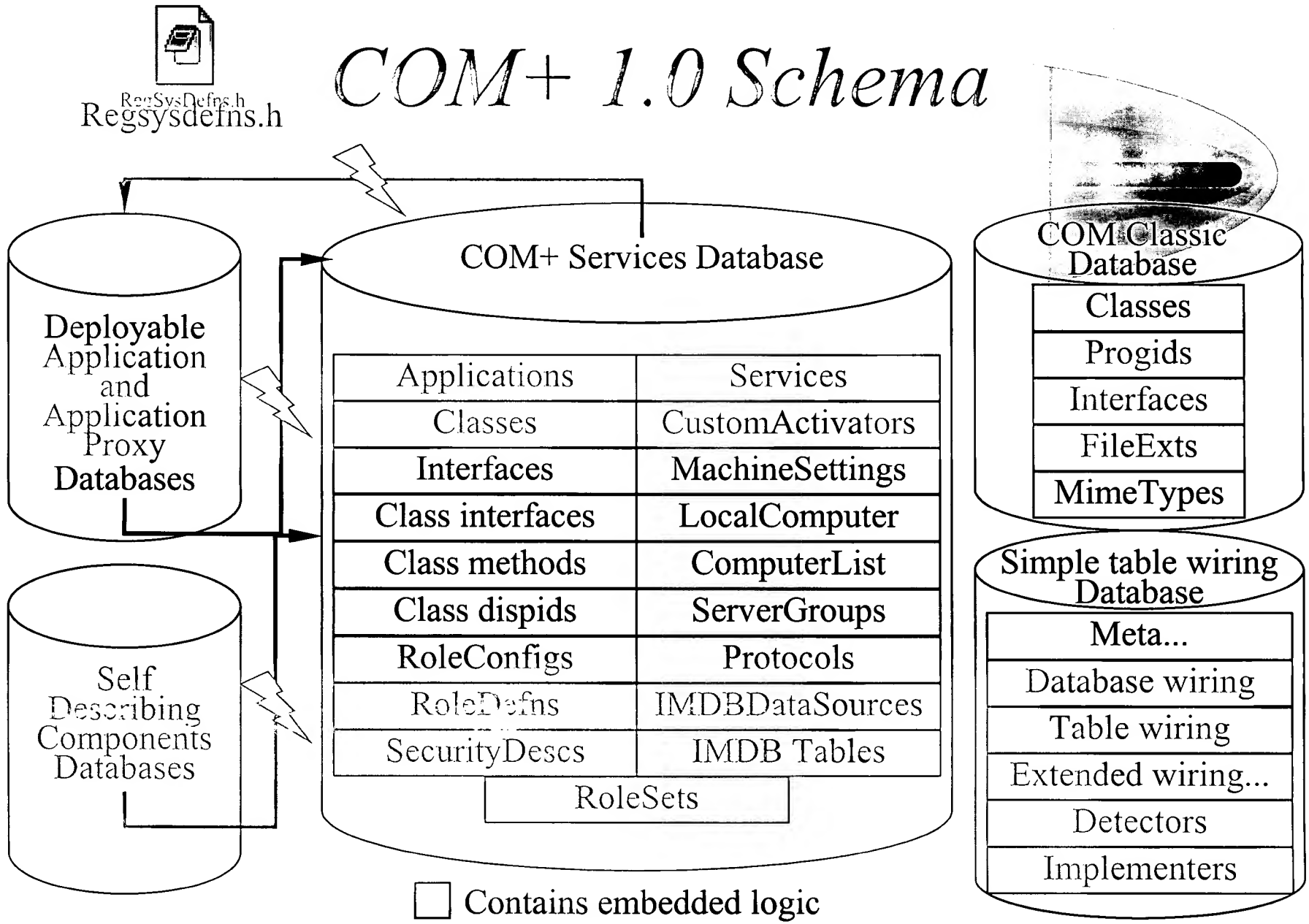
  Give the big picture of the catalog architecture followed by an introduction to each of its major design elements, to give an understanding of how the catalog works.

- Key Philosophies and Discussion:

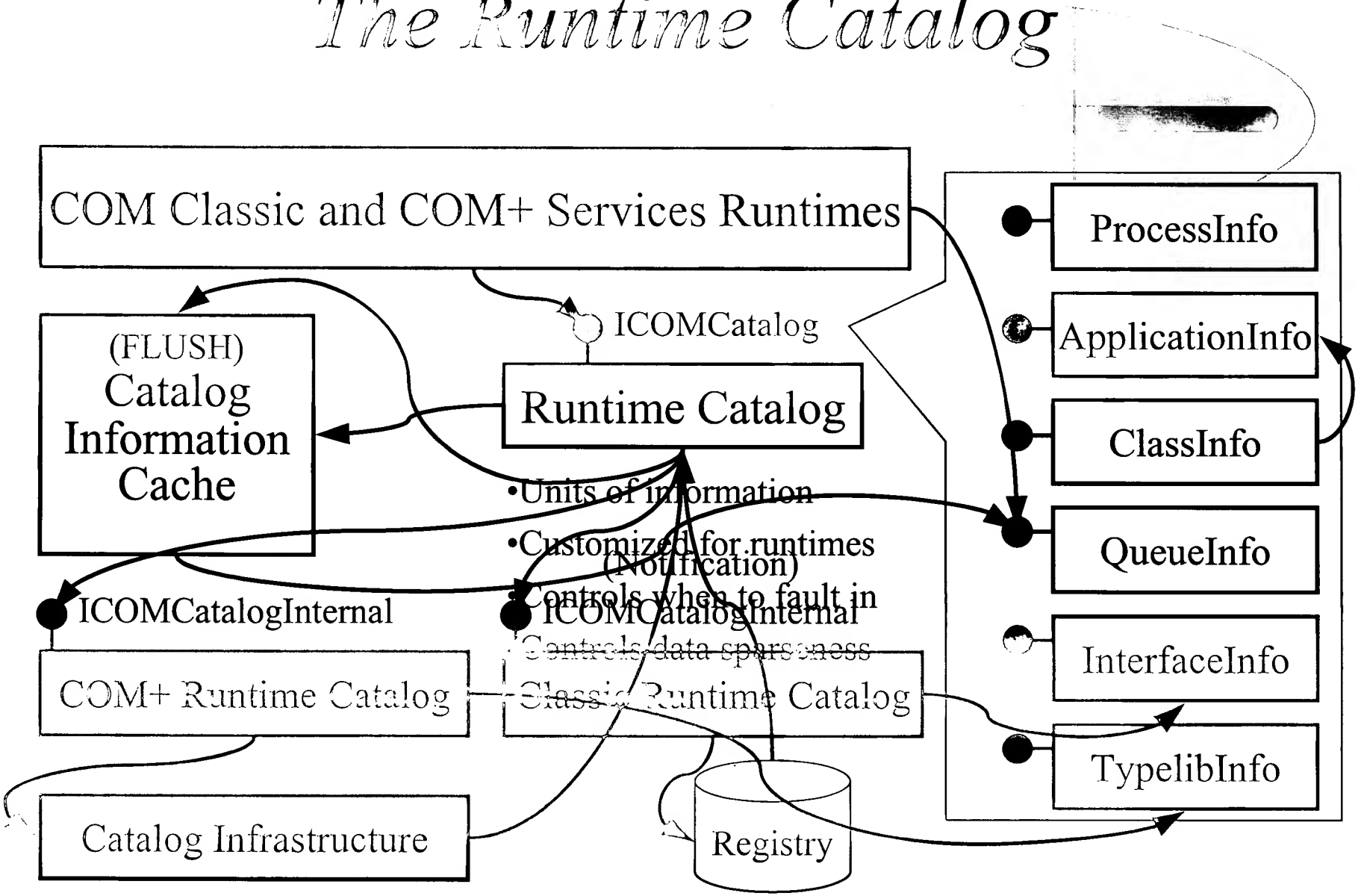
  Summarize the key philosophies that have emerged over the last 3 years of shipping the catalog, to initiate discussion on current and future design topics and issues.

The Catalog Architecture




RegSysDefns.h
Regsysdefns.h
COM+ 1.0 Schema
COM+ Services Database
Deployable Application and Application Proxy Databases
Self Describing Components Databases
Applications
Services
Classes
CustomActivators
Interfaces
MachineSettings
Class interfaces
LocalComputer
Class methods
ComputerList
Class dispids
ServerGroups
RoleConfigs
Protocols
RoleDefns
IMDBDataSources
SecurityDescs
IMDB Tables
RoleSets
COM Classic Database
Classes
Progids
Interfaces
FileExts
MimeTypes
Simple table wiring Database
Meta...
Database wiring
Table wiring
Extended wiring...
Detectors
Implementers
Contains embedded logic

The Runtime Catalog



stable.IDL

Simple Tables

ISimple*TableDispenser
(1 method)

ISimpleTableRead
(9 methods; 6 required)

1 Datastore (populate)
4 Row navigation
1 Get column data
2 Meta information
1 Clone

caller data

Byref data exchange

Read cache

Good minimums:

8 words+meta+data

Write cache

ISimpleTableWrite
(9 methods)

1 Datastore (update)
2 Row navigation
4 Row changes
2 Column data get / set

Datastore interaction

ISimpleTableControl
(12 methods)

Few caches

Many datastores

GetTable

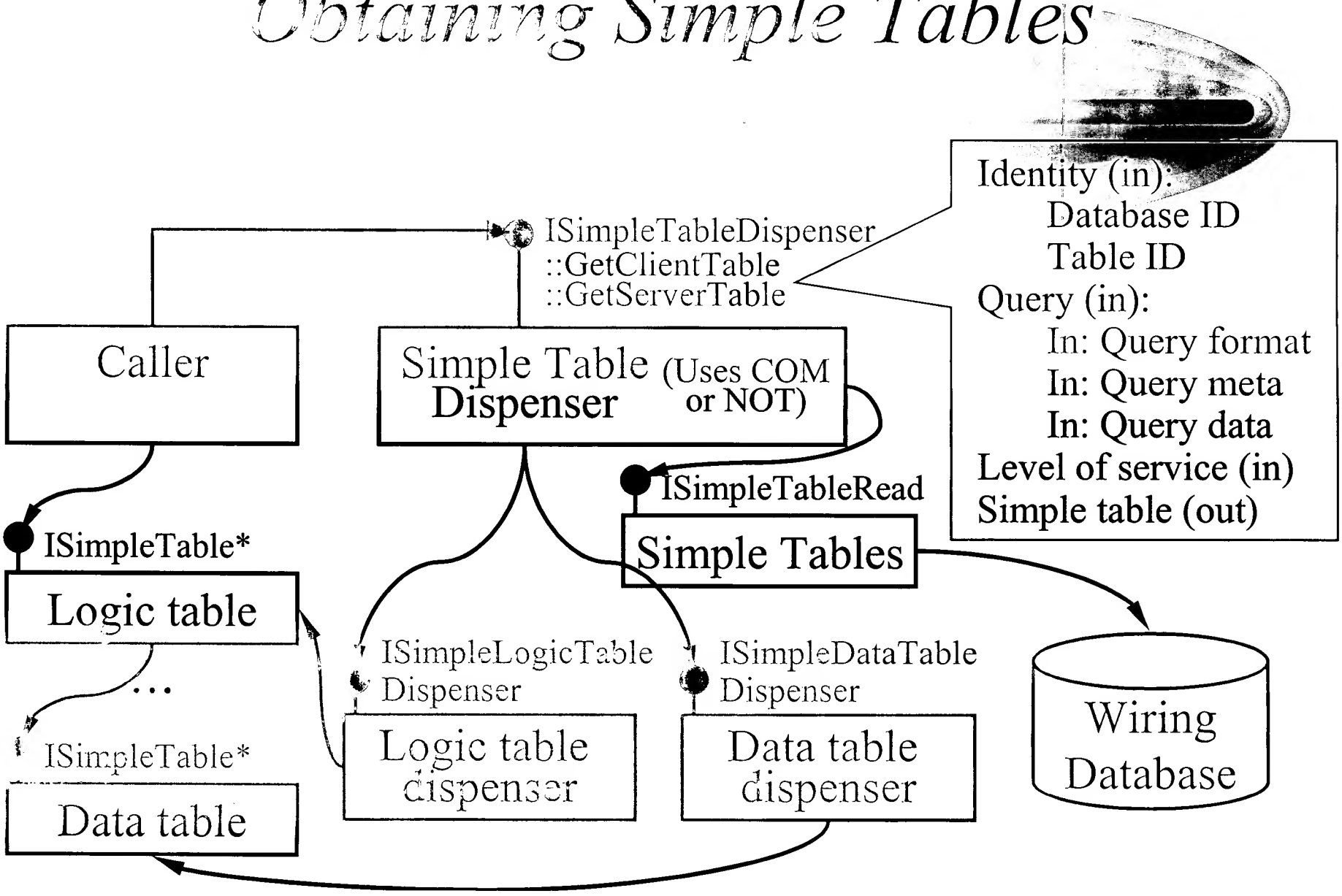
PopulateCache

UpdateStore

Datastore

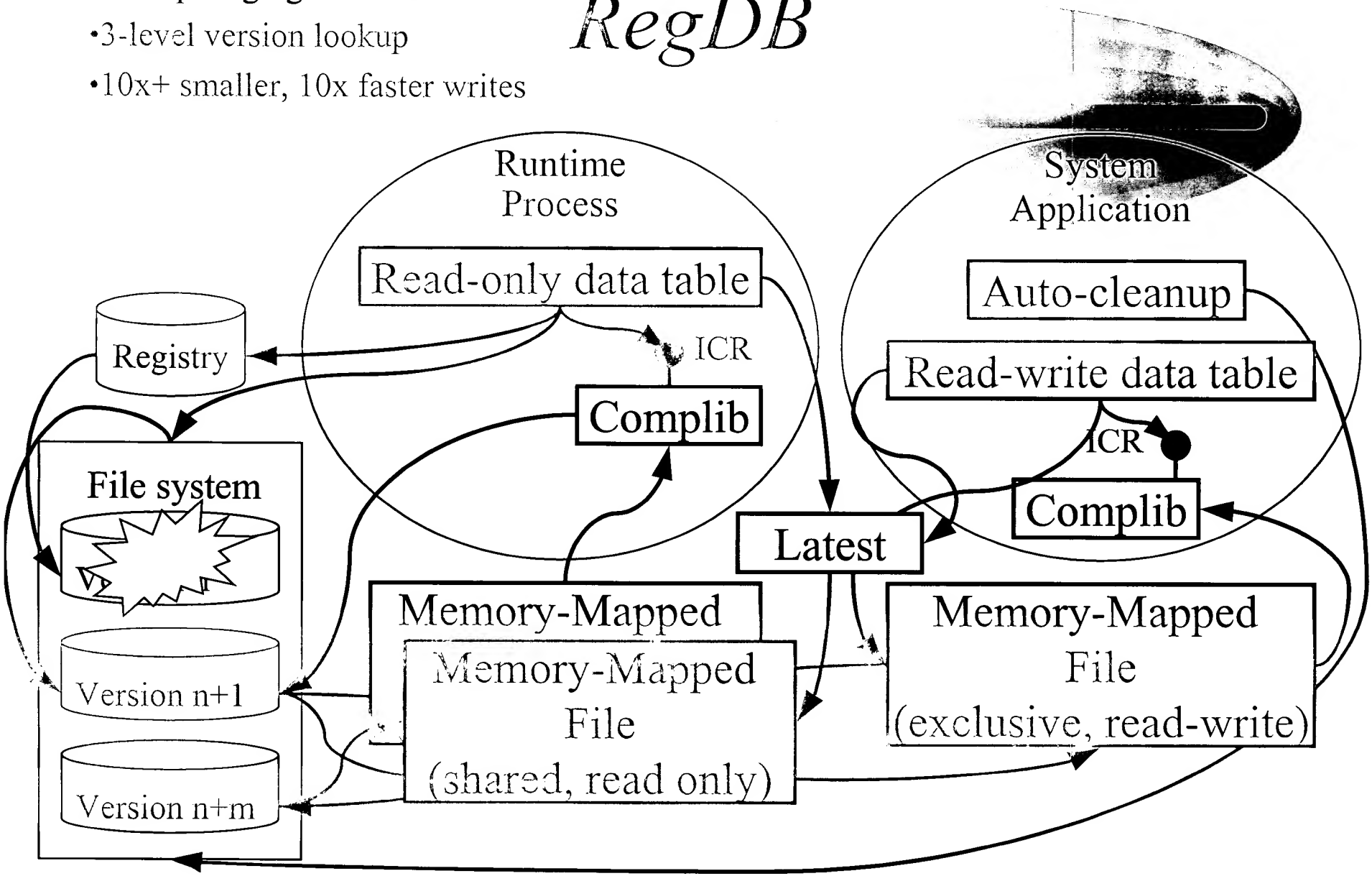
5 Cache manipulations
3 Write row methods
3 Detailed error
1 Marshalling

Obtaining Simple Tables



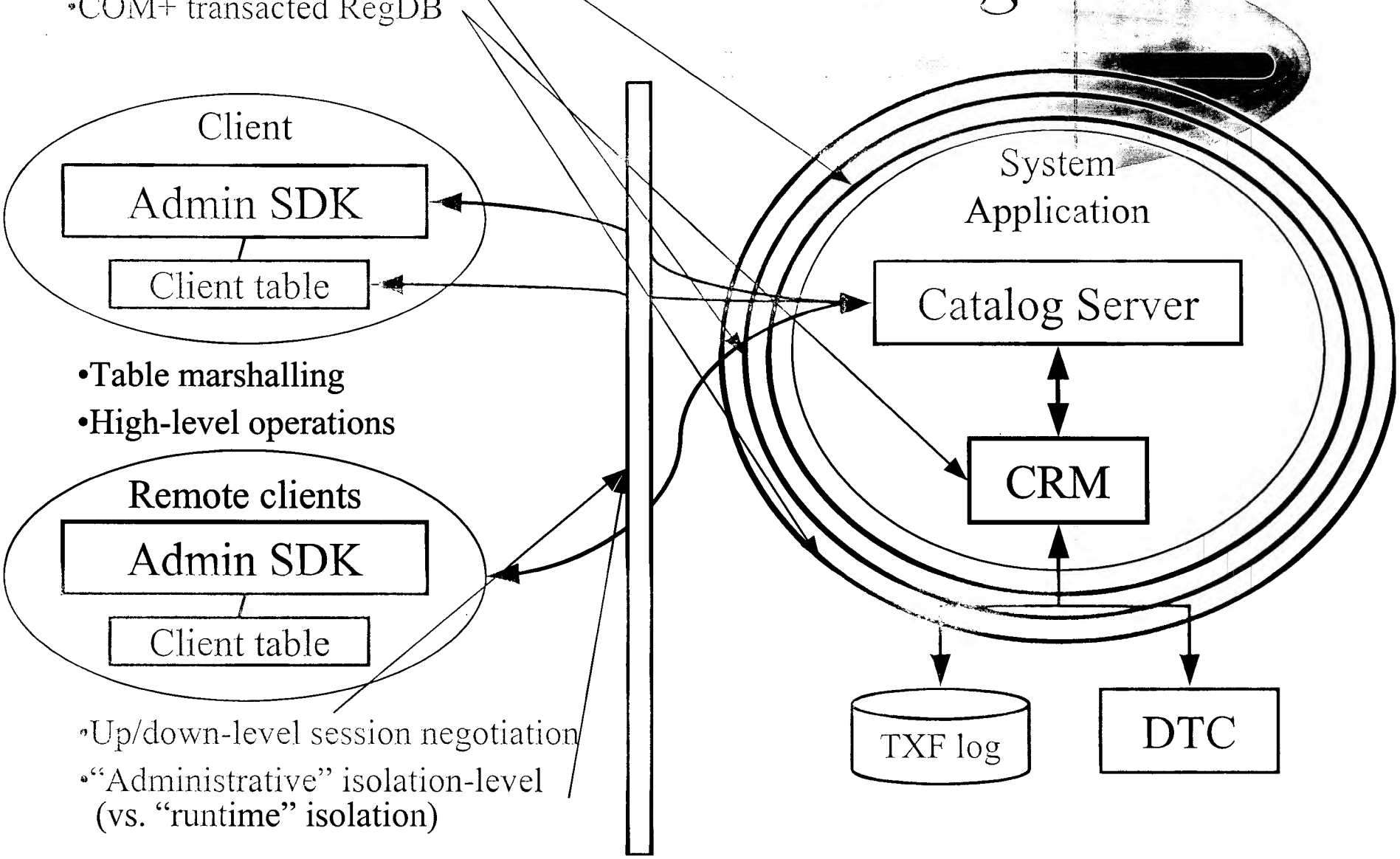
- Read-only, read-write
- Component library-based
- Multiple aging versions
- 3-level version lookup
- 10x+ smaller, 10x faster writes

RegDB



The Catalog Server

- Hosted by COM+ dllhost
- Secured by COM+ roles
- COM+ transacted RegDB



- Table marshalling
- High-level operations

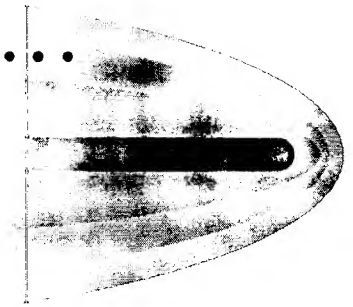
Remote clients

Admin SDK

Client table

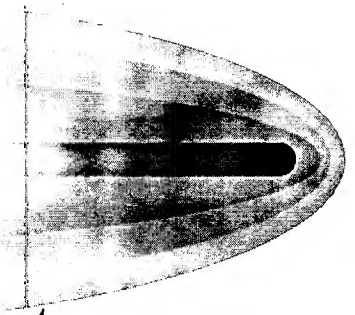
- Up/down-level session negotiation
- "Administrative" isolation-level (vs. "runtime" isolation)

Other Design Elements...



- Shipping in NT5:
  - Application deployment…
  - Remote administration…
  - Replication (NT Clusters / IIS server farms)...
- Future:
  - "Enterprise"-level application management…
  - Side-by-side versioning...
  - Application-relative configuration...
  - Just-in-time configuration…
  - "File operation" application install and uninstall...
  - "Anyone" extensibility...

Slide Outline



- Features: Past and Present:

  Talk about catalog features, both internal and customer visible, with a historical perspective, to give an understanding of the problems the catalog solves.

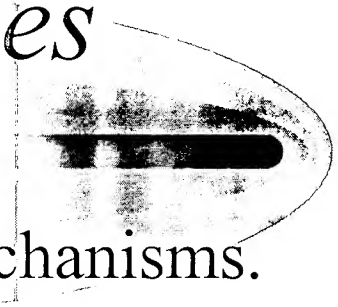
- Design Elements:

  Give the big picture of the catalog architecture followed by an introduction to each of its major design elements, to give an understanding of how the catalog works.

- Key Philosophies and Discussion:

  Summarize the key philosophies that have emerged over the last 3 years of shipping the catalog, to initiate discussion on current and future design topics and issues.

Key Catalog Philosophies



- Cleanly separate usage patterns / storage mechanisms.
- Serve runtime and configuration-time scenarios as distinct, important priorities.
- Recognize virtual meta data and physical configuration data as distinct and important.
- Automate configuration / simplify management.
- Acknowledge and plan for distributed applications.
- Design requirements must include perf *and* scaling.
- Be simple enough to support "anyone" extensibility.
- Dogfood useful services.
- Aggressively deliver and ship.

End of slides

Microsoft News
Products & Issues
Consumer News
International News
Legal News
Events

Microsoft Executives
Exec Bios/Speeches
Board of Directors
Bill Gates Web Site
Executive E-Mail

Other Corporate Info
Investor Relations
Analyst Relations
Fast Facts About Microsoft
Image Gallery
Microsoft Research
Essays on Technology
Community Affairs

Archives by Month
Press Releases
Top Stories

Company Focuses on Quality and Customer Feedback

REDMOND, Wash. - Aug. 18, 1998 - - Microsoft Corp. today announced the release of the second beta version of the Microsoft® Windows NT® Workstation 5.0, Windows NT Server 5.0, and Windows NT Server 5.0 Enterprise Edition operating systems. Beta 2 is the first release intended specifically for IT professionals to begin testing Windows NT 5.0 in their organizations. Beta code will be distributed to more than 45,000 technical beta sites, 25,000 channel partners and 200,000 developers worldwide via the Microsoft Developer Network.

Windows NT Workstation 5.0 is the premier desktop operating system for businesses of all sizes and is designed to replace Windows® 95 as the standard business desktop. Microsoft Windows NT Server 5.0, the next release of the best-selling server operating system, is the only multipurpose server operating system designed to connect, run and manage every part of the digital business. When combined, Windows NT Workstation 5.0 and Windows NT Server 5.0 provide a platform that lowers the total cost of ownership, enables a new generation of distributed applications and provides the infrastructure for companies to build their digital nervous systems.

"The Windows NT 5.0 platform will have a significant impact on how organizations reduce overall IT costs and build the infrastructure to enable distributed business applications," said Jim Allchin, senior vice president, personal and business systems group at Microsoft. "Our No. 1 goal for Windows NT 5.0 is quality. To date, the product has already undergone extensive testing by more than 100,000 beta testers, ISVs, OEMs and IHVs. Beta 2 will enable an even broader level of testing across a wider range of scenarios and features in the product."

Windows NT Workstation 5.0

Windows NT Workstation 5.0 is the next-generation desktop operating system that maximizes the power of the PC as a robust productivity tool for every business user. Windows NT Workstation 5.0 will be the compelling upgrade for the standard Windows® 95 operating system business desktop. End users and IT managers will benefit from moving to

Windows NT Workstation 5.0 with or without Windows NT Server. The product will deliver the following:

- **Easiest-to-use business desktop yet.** Windows NT Workstation 5.0 will offer a simpler and more intelligent user interface, including improved help and search capabilities, richer error messages and the best Internet experience. In addition, it will be significantly easier for users to configure their system with a comprehensive set of intuitive wizards.
- **The best of Windows 98 for business.** Windows NT Workstation 5.0 will offer key features found in Windows 98 that are important for businesses today, such as support for laptops via Plug and Play and power management. Also included is support for DVD and the hundreds of Universal Serial Bus devices introduced with Windows 98, such as scanners, printers and mouse products. Finally, Windows NT Workstation 5.0 will enable a new generation of hardware, such as digital cameras and recorders, by supporting fast communication technologies such as IEEE 1394.
- **Traditional Windows NT power.** Windows NT Workstation 5.0 continues to advance its core benefits of reliability, security and performance. It will reduce the number of planned and unplanned system reboots, deliver advanced security via richer permissions and expanded administration controls, and provide a more responsive computing experience for the standard business desktop. Windows NT Workstation 5.0 builds on the Windows NT architecture, providing a smooth upgrade for current users of Windows NT Workstation 4.0.
- **Lower total cost of ownership.** Windows NT Workstation 5.0 will offer lower overall management costs by making it easier to deploy, administer and support the business desktop. When combined with Windows NT Server 5.0, Windows NT Workstation 5.0 provides the client-side support for the new IntelliMirror™ management technologies, which provide an even greater reduction in total cost of ownership.

Windows NT Server 5.0

Microsoft Windows NT Server 5.0 builds on the strengths of Windows NT Server 4.0 by providing a platform that is faster, more reliable and easier to manage. Most important, it delivers a comprehensive set of distributed infrastructure services based on the Active Directory directory service, the first multipurpose directory service that is scalable, built from the ground up using Internet-standard technologies, and fully integrated at the operating system level. Although Windows NT Server 5.0 offers significant new functionality, the product is designed for modular deployment to allow customers to take advantage of this new technology at their own pace. Upon final release, Windows NT Server 5.0 will deliver the following benefits:

- **Increased manageability.** Windows NT Server 5.0 helps deliver a significantly lower total cost of ownership by centrally managing servers, desktops and networks through centralized policy-based administration provided by Active Directory and IntelliMirror, and powerful new remote administration tools.
- **Most comprehensive.** Windows NT Server 5.0 integrates best-of-breed file, print, Web, application, management, networking and communications, terminal emulation, and streaming media services to make it easier to implement powerful new business solutions.
- **Best long-term investment.** Windows NT Server 5.0 uses companies' existing investments in Windows, UNIX and NetWare, takes advantage of the latest hardware and networking advancements as they become available, and builds on the continued industry investment of more than 4,000 Windows NT-based applications and 170,000 Windows NT Server certified professionals.
- **Higher scalability and availability.** Windows NT Server 5.0 Enterprise Edition offers enhanced functionality, ease of configuration, scalability and availability to meet the needs of the most demanding mission-critical environments. Beta 2 introduces improvements in scalability through further symmetric multiprocessing (SMP) optimization, support for physical memories greater than 4 GB physical memories (up to 64 GB on some systems), clustering enhancements, and high-performance sorting for commercial data-warehouse and data-mart applications.

"We will be deploying Windows NT 5.0 beta 2 in production environments," said Ned Studt, senior engineer, research and development, IKON Corp., a unit of IKON Office Solutions Inc. "Beta 2 provides both the features and level of quality required for us to begin massive

deployments of the beta across the company to help us reduce the total cost of ownership and begin development of critical line-of-business applications."

Product Milestones

Microsoft is committed to delivering a high-quality product. Beta 3, the next major milestone in the development process, will be available once the company has received and incorporated feedback from beta 2 customers. The final release date for Windows NT 5.0 has not been set. System requirements, pricing and packaging also have not yet been determined. For more information on Windows NT 5.0, please see http://www.microsoft.com/windows/.

Founded in 1975, Microsoft (Nasdaq "MSFT") is the worldwide leader in software for personal computers. The company offers a wide range of products and services for business and personal use, each designed with the mission of making it easier and more enjoyable for people to take advantage of the full power of personal computing.

Note to editors: If you are interested in viewing additional information on Microsoft, please visit the Microsoft Web page at http://www.microsoft.com/presspass/ on Microsoft's corporate information pages.